Clube Delphi

Mini-Curso
Padrões de Projeto

Delphi Prism e ASP.NET
Reaproveite seus conhecimentos Win32 / Desktop para criar aplicações Web – Parte 5

Ano 9 :: Edição 122 :: R$ 11,90

DevMedia group

Lazarus e Free Pascal

Desenvolvimento RAD com ferramentas free e open source

Brinde

Comprando esta edição ganhe 2 vídeo-aulas:

- Acesso a dados no Delphi Prism - SqlCommand
- Acesso a dados no Delphi Prism – XML e SQL Server

Quick Update
Instalações descomplicadas com Inno Setup e ISTool

Boas Práticas
Projetando softwares Delphi da forma correta

ISSN 1517990-7

Dificuldades em encontrar conteúdo sobre .Net para iniciantes?

A DevMedia simplifica pra você!

easy .net magazine

Easy .Net Magazine:

A primeira revista digital feita especialmente para quem está dando os primeiros passos em .Net!

Você encontrará exemplos e modelos fáceis e práticos, noções introdutórias e conceitos básicos.

Conheça as vantagens:

- Vídeos e apresentações "embutidos" nas páginas;
- Ilustrações, diagramas e modelos apresentados em PowerPoint;
- Entrevistas em vídeo;
- Códigos-fonte que podem ser copiados com CTRL+C/CTRL+V.

A Easy.Net não é feita em PDF ou Flash!
Ela utiliza um aplicativo 100% online.

DevMedia group

Acesse agora a edição Nº 1 totalmente grátis e comprove

www.devmedia.com.br/easynet

Easy.Net Magazine, seu curso inicial de .Net!

Sumário

Quick Update

08 – Inno Setup e ISTool
[*Daniel Sobrinho Laporte*]

Delphi — Tutorial

11 – Lazarus e FreePascal
Conheça o IDE Lazarus e o compilador FreePascal, uma ótima alternativa Open Source – Parte 1
[*Daniel Simões Almeida*]

Boas Práticas — Boas Práticas – Eng. de software

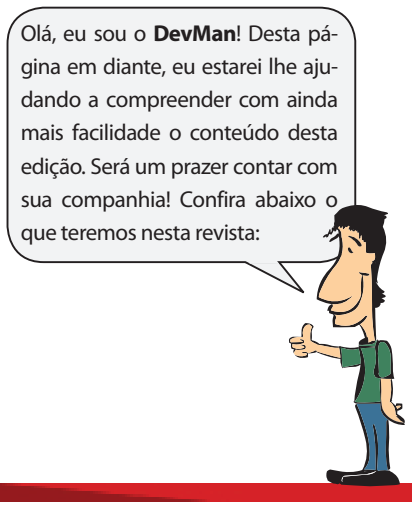
22 – Criando um grande software
Aprenda como criar um software apreciado por todos
[*Rafael Stavarengo*]

Boas Práticas — Boas Práticas

34 – Design Patterns - Parte 1
Abstract Factory e Factory Methods
[*Vitor Luiz Rubio*]

Easy Delphi — Easy Delphi - Mini-Curso – Tutorial – Web – Novidades

44 – Delphi Prism e ASP.NET
Reaproveite seus conhecimentos Win32 / Desktop para criar aplicações Web – Parte 5
[*Guinther Pauli*]



Brinde na web desta edição - 2 vídeos! Confira:

2 Vídeos

• **SqlCommand e ExecuteScalar: Retornando um Object - Curso Completo Delphi Prism - Desenvolvimento Web**
http://www.devmedia.com.br/post-17602-SqlCommand-e-ExecuteScalar--Retornando-um-Object--Curso-Completo-Delphi-Prism-Desenvolvimento-Web-Parte-60.html

• **DataSet e XML com Banco de dados SQL SERVER - Curso Completo Delphi Prism - Desenvolvimento Web**
http://www.devmedia.com.br/post-17680-DataSet-e-XML-com-Banco-de-dados-SQL-SERVER-Curso-Completo-Delphi-Prism-Desenvolvimento-Web-Parte-62.html

Para acessar os vídeos utilize os seguintes dados
Login: **DVM.PLS** *Senha:* **T4Y1**

Gostou das vídeo aulas? O portal *www.devmedia.com.br* possui mais de **2 mil vídeo aulas** e **dezenas de cursos online** sobre desenvolvimento de software! Agora você pode comprar as vídeo aulas que preferir e fazer sua própria combinação de vídeos! Saiba mais em *www.devmedia.com.br/creditos*

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Para votar, você vai precisar do código e senha de banca desta edição, que é:
Login: **DVM.PLS** Senha: **T4Y1**

Clube Delphi

Ano 9 - 122ª Edição 2010 - ISSN 1517990-7 Impresso no Brasil

Corpo Editorial

Editor Geral
Guinther Pauli
guinther@devmedia.com.br

Editor Técnico
Paulo Quicoli
pauloquicoli@devmedia.com.br

Daniel Laporte
daniel.sobrinho@devmedia.com.br

Rodrigo Carreiro Mourão
rodrigocarreiro@rmfactory.com.br

Equipe Editorial
Vitor Luiz Rubio
vitorrubio@gmail.com

Rafael Stavarengo
faelsta@gmail.com

Capa e Diagramação
Romulo Araujo
romulo@devmedia.com.br

Coordenação Geral
Daniella Costa - daniella@devmedia.com.br

Revisão e Supervisão
Thiago Vincenzo - thiago.v.ciancio@devmedia.com.br

Distribuição
Fernando Chinaglia Dist. S/A
Rua Teodoro da Silva, 907
Grajaú - RJ - 206563-900

Atendimento ao Leitor

A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

Cristiany Queiroz – Atendimento ao Leitor
www.devmedia.com.br/mancad
(21) 3382-5038

Kaline Dolabella – Gerente de Marketing e Atendimento
kalined@terra.com.br
(21) 3382-5038

Publicidade

Para informações sobre veiculação de anúncio na revista ou no site e para fechar parcerias ou ações específicas de marketing com a DevMedia, entre em contato com:

Kaline Dolabella
publicidade@devmedia.com.br

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!
Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Guinther Pauli - Editor da Revista
guinther@devmedia.com.br

EDITORIAL

O Lazarus e Free Pascal são ferramentas livre e open source para o desenvolvimento de aplicações para várias plataformas como Windows, Linux, WinCE e MacOS. O IDE do Lazarus é praticamente idêntico ao IDE do Delphi 7, com recursos de form designer, debugger, editor de código, inclusive com opções que só vemos nas versões mais atuais do RAD Studio. A forma como você desenvolve aplicações é praticamente idêntica ao Delphi Win32 tradicional, baseado nas técnicas RAD e Drag & Drop. O grande ponto a ser destacado é que este projeto é livre, você pode desenvolver aplicações de forma RAD com excelentes recursos para plataformas nativas sem gastar um centavo. Além disso, o projeto é desenvolvido com a colaboração de vários renomados experts na área de desenvolvimento Delphi, que garantem um nível de maturidade e seriedade muito grande ao produto. E já existem, inclusive, muitos softwares em produção que utilizam estas ferramentas, alguns inclusive migrados de soluções Delphi tradicionais. O Daniel Simões, que participa do desenvolvimento do Lazarus (e que também é idealizador e principal mantenedor do Projeto ACBr), apresenta nesta edição um artigo sobre estas ferramentas, contando também sua experiência prática ao migrar um projeto real.

Em nossa sessão Boas Práticas o Rafael Stavarengo, usando uma metodologia muito interessante, apresenta algumas técnicas de engenharia de software direcionadas ao público Delphi para projetar de forma correta grandes sistemas. O autor cria um cenário onde apresenta um software a ser modelado, passando pela UML, análise e implementação. Simula também um problema na aplicação, que é semelhante ao que vemos no dia-a-dia, mostrando de uma forma simples e inteligente (e porque não dizer, irônica) como podemos projetar sistemas para que possam estar adaptados a mudanças e livres de bugs. Usando claro, para isso, excelentes práticas de orientação a objetos e Design Patterns.

Design Patterns que por sinal é um tema constante aqui na ClubeDelphi e destaque na matéria do Vitor Rubio. Ele apresenta um dos padrões mais conhecidos, do grupo de patterns criacionais, as fábricas de objetos (padrões como Abstract Factory e Factory Method). De praxe, ele vai mostrar como usar este padrão em um exemplo da vida real.

Na sessão Easy, continuo a nossa série que está mostrando como você pode desenvolver aplicações para Web com ASP.NET de forma muito semelhante ao que está acostumado a fazer em Delphi Win32, onde destaco novamente vários comparativos entre as plataformas .NET e Win32. Este mês, você vai conhecer a fundo o ADO.NET e também vai criar um carrinho de compras em nossa loja virtual.

Boa Leitura!

Guinther Pauli

guinther@devmedia.com.br

Certificado Delphi 3,5,6,7,2005,2006,Web,Kylix

Microsoft Certified: MCP,MCAD,MCSD.NET,MCTS,MCPD (ASP.NET,C#)

guintherpauli.blogspot.com - twitter.com/guintherpauli

ClubeDelphi PLUS

A revista ClubeDelphi é parte integrante da assinatura ClubeDelphi PLUS.
Para mais informações sobre o pacote PLUS, acesse:
http://www.devmedia.com.br/clubedelphi/portal.asp

DevMedia group

Existem coisas que não conseguimos ficar sem!

...só pra lembrar, sua assinatura está acabando!

Renove Já!

www.devmedia.com.br/renovacao

Perguntas e Respostas

Hackeando o DBNavigator para inserir imagens

Pergunta respondidas por:
Daniel Sobrinho Laporte

Esta é para os fissurados em visual. Quem faz uso de componentes Data-Aware com certeza deve utilizar muito o componente DBNavigator, que como sabemos, realiza as tarefas de manipulação de um DataSet de uma maneira muito mais simplificada. Porém aqui há um ponto que muitos desenvolvedores às vezes optam por desenvolver uma solução diferente, adotando SpeedButtons, BitBtns, componentes de terceiros ou até mesmo criando seus próprios componentes, no intuito de inserir imagens em um DBNavigator. Nesta sessão irei demonstrar como exibir imagens em um DBNavigator sem a necessidade de utilizar componentes de terceiros, criar componentes descendentes ou mesmo adotar as outras soluções que foram mencionadas anteriormente. A grande vantagem é que podemos separar isto em uma biblioteca de funções, por exemplo, ou alguma classe gráfica específica e reaproveitar este código para todo o seu projeto. Vamos ao exemplo prático.

Primeiramente vá em File>New>VCL Forms Application para criar uma nova aplicação. Adicione ao novo formulário um DBNavigator e um componente ImageList. Veja o layout na **Figura 1.**

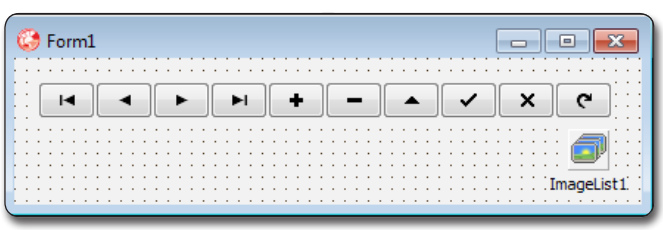


Figura 1. Layout da aplicação

Se você observar no Object Inspector o componente DBNavigator verá que o mesmo não possui uma propriedade para linkar com o ImageList, desta forma, necessitaremos "hackear" a classe do DBNavigator. Aqui uso o termo "hackear" pela seguinte explicação: Caso você declare um objeto de uma determinada classe, é possível acessar seus métodos internos, algo que você não conseguiria fazer utilizando apenas o componente DBNavigator. Desta forma, iremos criar uma nova Unit para simularmos uma biblioteca de funções externa que poderá ser reutilizada em outros projetos. Siga em File>New>Unit. Salve a nova Unit com o nome de "untFuncoes.pas", o formulário principal e o projeto não necessitarão de modificações, portanto, deixe-os nomeados como "Unit1.pas" e "Project1.dproj" respectivamente.

O primeiro passo a se fazer será "hackear" a classe. Abra a untFuncoes e abaixo da cláusula interface declare uma classe chamada THackDBNavigator conforme a seguir:

```
//Adicione ao uses as Units DBCtrls, Controls e Graphics;
 type
  THackDBNavigator = class(TDBNavigator);
```

Com isso já é possível acessar os métodos internos da classe do DBNavigator fazendo para isso apenas um Type Cast. O segundo passo será declarar a classe que possuirá o método que insere as imagens no DBNavigator. Para isso utilizaremos uma classe estática, conforme mostrado a seguir:

```
type
  TGraphicUtil = class
public
  class procedure InsertGlyphsDBNavigator(ImageList: TImageList;
    DBNavigator: TDbNavigator);
end;
```

Pressione as teclas Ctrl+Shift+C para que o corpo do método seja automaticamente implementado. A seguir codifique-o conforme a **Listagem 1**.

Nota do DevMan

A palavra reservada class antes da assinatura do método faz com que o método seja estático, ou seja, dispensa a criação da classe, sendo possível já realizar a chamada de forma direta, pois está já se encontrará instanciada.

Listagem 1. Implementação do método InsertGlyphsDBNavigator

```
class procedure TGraphicUtil.InsertGlyphsDBNavigator(ImageList: TImageList;
 DBNavigator: TDbNavigator);
begin
 THackDBNavigator(DBNavigator).Buttons[nbInsert].Glyph.PixelFormat := pf32bit;
 THackDBNavigator(DBNavigator).Buttons[nbEdit].Glyph.PixelFormat  := pf32bit;
 THackDBNavigator(DBNavigator).Buttons[nbDelete].Glyph.PixelFormat := pf32bit;
 THackDBNavigator(DBNavigator).Buttons[nbPost].Glyph.PixelFormat  := pf32bit;
 THackDBNavigator(DBNavigator).Buttons[nbCancel].Glyph.PixelFormat := pf32bit;
 THackDBNavigator(DBNavigator).Buttons[nbFirst].Glyph.PixelFormat  := pf32bit;
 THackDBNavigator(DBNavigator).Buttons[nbPrior].Glyph.PixelFormat  := pf32bit;
 THackDBNavigator(DBNavigator).Buttons[nbNext].Glyph.PixelFormat  := pf32bit;
 THackDBNavigator(DBNavigator).Buttons[nbLast].Glyph.PixelFormat  := pf32bit;
 ImageList.GetBitmap(0, THackDBNavigator(DBNavigator).Buttons[nbFirst].Glyph );
 ImageList.GetBitmap(1, THackDBNavigator(DBNavigator).Buttons[nbPrior].Glyph );
 ImageList.GetBitmap(2, THackDBNavigator(DBNavigator).Buttons[nbNext].Glyph );
 ImageList.GetBitmap(3, THackDBNavigator(DBNavigator).Buttons[nbLast].Glyph );
 ImageList.GetBitmap(4, THackDBNavigator(DBNavigator).Buttons[nbInsert].Glyph );
 ImageList.GetBitmap(5, THackDBNavigator(DBNavigator).Buttons[nbDelete].Glyph );
 ImageList.GetBitmap(6, THackDBNavigator(DBNavigator).Buttons[nbEdit].Glyph );
 ImageList.GetBitmap(7, THackDBNavigator(DBNavigator).Buttons[nbPost].Glyph );
 ImageList.GetBitmap(8, THackDBNavigator(DBNavigator).Buttons[nbCancel].Glyph );
 ImageList.GetBitmap(9, THackDBNavigator(DBNavigator).Buttons[nbRefresh].Glyph);
end;
```

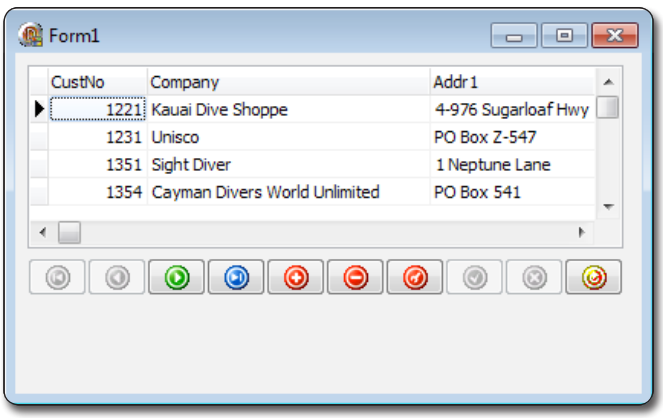


Figura 2. Layout final com as imagens no DBNavigator

Perceba no código da **Listagem 1** que recebemos um DBNavigator como parâmetro na procedure, porém, tentando acessar a propriedade Buttons[NavigateButton].Glyph você notará que não é permitido. Porém através do Type Cast da classe THackDBNavigator nós conseguiremos acessar normalmente estas propriedades. Primeiramente modificamos o PixelFormat das imagens, passando-as para 32 bits, assim faremos com que tenhamos uma aparência melhor nos Glyphs. Em seguida utilizamos o método GetBitmap (que retorna uma imagem) para preencher a propriedade Glyph dos Buttons. Note que o primeiro parâmetro de 0 a 9 será a posição da imagem dentro do ImageList.

Nota

O arquivo customer.cds se encontra no diretório da aplicação de exemplo, para download.

O próximo passo é adicionarmos as imagens a um ImageList em nosso formulário principal e realizar a chamada da rotina no formulário. Para um teste das funcionalidades vamos adicionar uma DBGrid, um ClientDataSet, um DataSource. Ligue a DBGrid e o DBNavigator ao DataSource e este ligue no ClientDataSet. Para carregar o nosso ClientDataSet com informações utilizaremos um arquivo XML qualquer, deste modo, clique de direita no ClientDataSet e selecione a opção "Load from MyBase table.." e escolha a opção ClientDataSet (cds). No meu caso utilizarei um dos arquivos de exemplo contidos no diretório de instalação. Carregaremos o arquivo "customer.cds" do diretório de exemplos.

Por último adicionaremos a chamada da rotina no evento OnShow do formulário:

```
//Adicione a unit untFuncoes ao uses do formulário
procedure TForm1.FormShow(Sender: TObject);
begin
  TGraphicUtil.InsertGlyphsDBNavigator(
   ImageList1,
   DBNavigator1);
end;
```

O resultado final você confere na **Figura 2**.

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!
Dê seu voto sobre este artigo, através do link:
www.devmedia.com.br/clubedelphi/feedback

Quick Update: Inno Setup e ISTool

Daniel Sobrinho Laporte
daniel.sobrinho@devmedia.com.br
daniel.laporte@gmail.com
www.twitter.com/daniel_laporte
Colunista e editor técnico da Revista ClubeDelphi. Graduado em Sistemas de informação e pós-graduado em Administração de Bancos de Dados pela Universidade de Franca. É analista de sistemas e desenvolvedor Delphi desde 2005 desenvolvendo aplicações cliente/servidor e multicamadas.

Sempre buscamos em nossos projetos, além do desenvolvimento adequado, a melhor maneira de distribuir nossas aplicações. E é claro que dentre estas maneiras podemos desenvolver aplicativos para a criação do executável, separando as tarefas necessárias como instalação do servidor de banco de dados, configuração do banco de dados, execução de algum script de inicialização, registro de aplicações, ou registros no Windows, entre uma série de outras coisas. Muitos profissionais desenvolvem suas próprias ferramentas para realizarem estas tarefas ou utilizam um arquivo compactado para distribuir tudo que for necessário, porém, existe um velho conhecido nosso, que permite a criação e customização do seu software em um instalador, este nosso conhecido é chamado de Inno Setup.

O Inno Setup é uma ferramenta Open Source (desenvolvida em Delphi) que possui uma sintaxe de scripts para a criação de instaladores bem profissionais. É possível fazer praticamente tudo que deseja, desde que se conheça o funcionamento dos scripts, mas o que fazer se você não possui este conhecimento? Vamos apresentar aqui uma ferramenta que pode auxiliar qualquer desenvolvedor a criar seus executáveis de uma maneira muito prática e simples, sem exigir nenhum conhecimento do Inno Setup, conheceremos o ISTool.

O ISTool assim como Inno Setup, é um projeto Open Source e trata-se de uma espécie de add-in para o Inno Setup, proporcionando uma interface para o auxílio da criação dos Scripts do Inno Setup. Agora que já conhecemos em teoria ambas as ferramentas, vamos conhecê-las na prática.

Nota

O link para o download das ferramentas utilizadas no artigo encontra-se na sessão Links ao final do mesmo.

Primeiramente é necessário instalar o Inno Setup. Para quem ainda não o conhece, darei uma pequena introdução de como utilizá-lo, que vai garantir fazer o mesmo que definirmos no Inno sem conhecimento algum de scripts. Com o Inno instalado vamos criar um projeto básico, apenas com a intenção de ilustrar um exemplo de como distribuir seu software. Criaremos para facilitar o entendimento uma estrutura de pastas para manter os fontes do sistema, o executável gerado, o script (do Inno Setup) e as imagens, portanto crie sua estrutura conforme sugere a **Figura 1**.

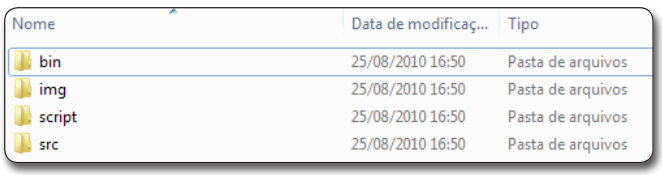


Figura 1. Estrutura de diretórios proposta

Para facilitar, deixei a estrutura da **Figura 1** dentro de uma pasta no diretório "C:\QuickUpdate". O primeiro passo que faremos é informar ao Delphi onde o executável deverá ser gerado. Abra seu Delphi 2010 e crie uma nova aplicação VCL Forms Win32. Não vamos alterar nada no formulário, pois o que nos interessa é só o executável, portanto, salve sua Unit e seu projeto na pasta src do diretório. O próximo passo é informar ao Delphi que o executável deve ser gerado na pasta bin. Para isso selecione o Projeto e clique de direita, selecionando a opção Options. Na nova janela selecione do lado esquerdo a opção Delphi Compiler e do lado direito, informe em Output directory o diretório bin, como mostra a **Figura 2**.

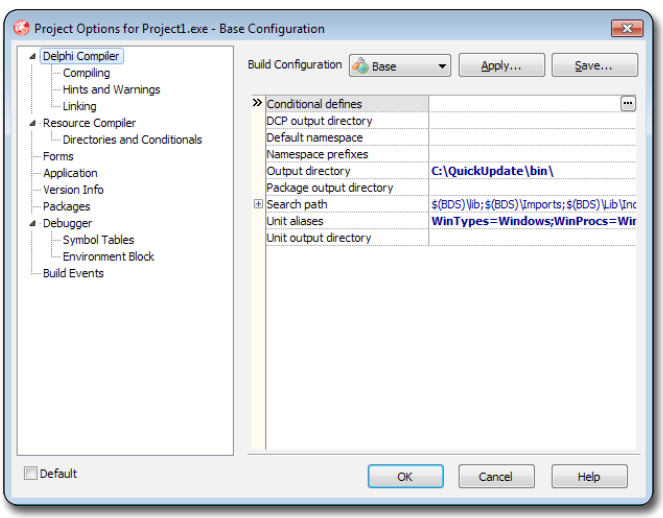


Figura 2. Local onde o executável será gerado

Após ter configurado, dê OK e compile seu projeto, abrindo a pasta você notará que o executável se encontra no diretório sugerido pela **Figura 2**. Vamos agora à parte do Inno Setup. Após instalado execute-o para criarmos um instalador de exemplo. A primeira janela que aparece nos exibe duas opções, a de criar um no script em branco, ou utilizar um Wizard (assistente) para a criação. Selecione a segunda opção para utilizarmos o caminho mais simples. Na primeira janela apenas dê Next, a próxima janela lhe pedirá as informações do nome do seu software, versão, distribuidor (nome de sua empresa) e seu website. Preencha as informações e novamente clique em Next. Em seguida teremos as opções de instalar o software em uma pasta específica, por padrão deixaremos definidas as opções padrão. Na janela da **Figura 3** devemos informar o nosso executável (do diretório bin) e todos os outros arquivos que você queira distribuir (midas.dll, fbclient.dll por exemplo), clique em Next novamente. As próximas telas configuram a criação das pastas e atalhos do programa, arquivos de licença e idioma respectivamente, apenas modifique o idioma para português e avance nas três telas. Por último é exibida a janela de regras de compilação, aonde podemos definir o nome do arquivo de saída, deixaremos setup conforme o valor inicial e clique em Next e Finish.

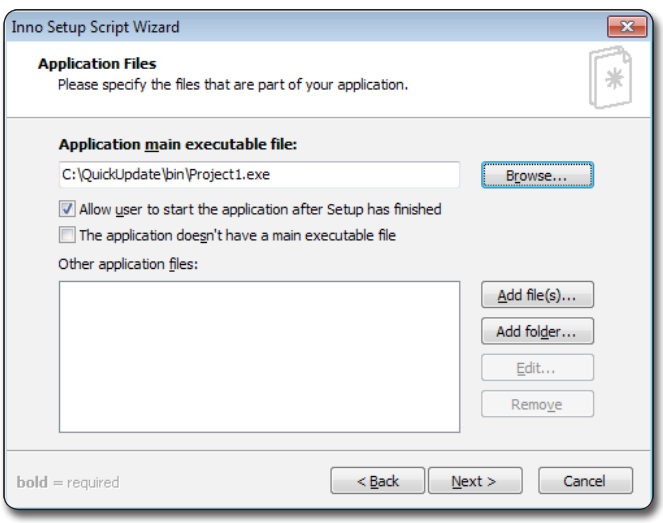


Figura 3. Inserindo o executável que será instalado

Veja o script que foi automaticamente criado na **Listagem 1**.

Note que o script possui um padrão a seguir, estudando este padrão e as regras é possível se aprofundar e criar inclusive validações através do mesmo. Agora que já temos um script, vamos salvá-lo na pasta script dentro do diretório proposto com o nome de "script.iss". Feito isso, abra o script gerado no ISTool. Observe na **Figura 4** que à esquerda possuímos um Treeview com todas as opções (que no script estão separadas pelos "[ ]"). Com isso, basta clicarmos de direita na área livre e inserir novos itens conforme a necessidade, ou mesmo criar os próprios scripts através do ISTool.

Acrescentando ou removendo itens, o ISTool automaticamente irá inserir a instrução ao script, desta forma facilita inclusive o aprendizado de sintaxe e compreensão dos padrões

Listagem 1. Script do Inno Setup

```
; Script generated by the Inno Setup Script Wizard.
; SEE THE DOCUMENTATION FOR DETAILS ON CREATING INNO SETUP SCRIPT FILES!

[Setup]
; NOTE: The value of AppId uniquely identifies this application.
; Do not use the same AppId value in installers for other applications.
; (To generate a new GUID, click Tools | Generate GUID inside the IDE.)
AppId={{CBF854E9-2E0E-444F-9EEB-AA5C45658DD8}
AppName=ClubeDelphi
AppVersion=1.5
;AppVerName=ClubeDelphi 1.5
AppPublisher=DevMedia
AppPublisherURL=http://www.devmedia.com.br/
AppSupportURL=http://www.devmedia.com.br/
AppUpdatesURL=http://www.devmedia.com.br/
DefaultDirName={pf}\ClubeDelphi
DefaultGroupName=ClubeDelphi
OutputBaseFilename=setup
Compression=lzma
SolidCompression=yes
[Languages]
Name: "brazilianportuguese"; MessagesFile: "compiler:Languages\
BrazilianPortuguese.isl"

[Tasks]
Name: "desktopicon"; Description: "{cm:CreateDesktopIcon}"; GroupDescription:
"{cm:AdditionalIcons}"; Flags: unchecked

[Files]
Source: "C:\QuickUpdate\bin\Project1.exe"; DestDir: "{app}"; Flags: ignoreversion
; NOTE: Don't use "Flags: ignoreversion" on any shared system files

[Icons]
Name: "{group}\ClubeDelphi"; Filename: "{app}\Project1.exe"
Name: "{commondesktop}\ClubeDelphi"; Filename: "{app}\Project1.exe"; Tasks:
desktopicon

[Run]
Filename: "{app}\Project1.exe"; Description: "{cm:LaunchProgram,ClubeDelphi}";
Flags: nowait postinstall skipifsilent
```

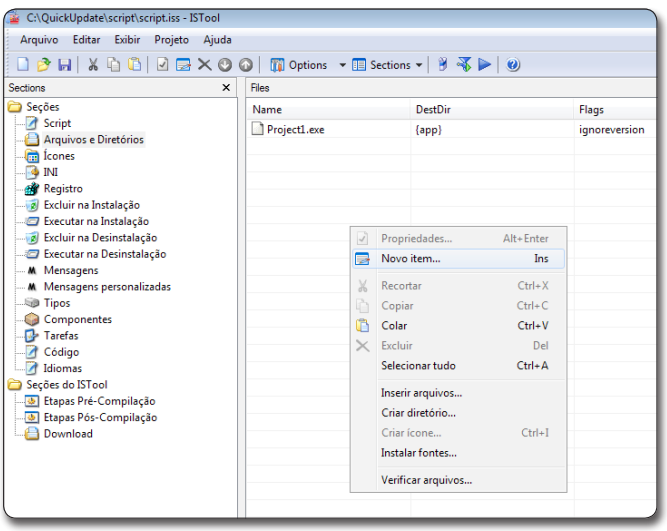


Figura 4. Utilizando o ISTool

de geração do código do instalador. Vale lembrar que o ISTool permite que todo o processo de compilação e geração do executável, seja gerado apenas por ele, desde que o Inno Setup esteja instalado, para isso basta utilizar o botão de compilação e instalação. Com isso, conhecemos mais uma ferramenta que pode tornar nossos softwares muito mais profissionais e dispensando qualquer tipo de complicação. Um grande abraço e até a próxima!

Inno Setup
http://www.jrsoftware.org/isinfo.php

ISTool
http://www.istool.org/

Blog do autor
http://twitter.com/daniel_laporte

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Lazarus e FreePascal

Conheça o IDE Lazarus e o compilador FreePascal, uma ótima alternativa Open Source – Parte 1

Daniel Simões Almeida

É no mínimo curioso ver um artigo sobre Lazarus em uma revista com o nome ClubeDelphi. Será que esse artigo está na revista correta? A resposta é SIM, pois Lazarus também é ObjectPascal. Você terá uma grande surpresa ao saber que o seu código Delphi é aproximadamente 90% compatível com o Lazarus (ou mais). Durante muito tempo, Delphi foi sinônimo de ObjectPascal. O poderoso compilador da Borland sempre ficou oculto por trás da fantástica IDE que é o Delphi.

Sempre que alguém falava sobre ObjectPascal, o Delphi nos vinha em mente. Porém hoje em dia, temos Delphi Prism, Delphi for PHP, e a relação de "um para um" do Delphi para com o ObjectPascal já não é mais uma realidade. A Borland fez um excelente trabalho, tornando o ObjectPascal uma das linguagens de maior sucesso de todos os tempos. O sucesso foi tanto, que despertou o interesse em alguns talentosos desenvolvedores em criar um clone do Delphi para outras plataformas, e assim surgiu o Lazarus, lá em meados de 1998.

Resumo DevMan

De que trata o artigo:

Neste artigo irei relatar pontos relevantes da minha experiência no processo de migração do IDE Delphi 7 para Lazarus/FreePascal. Procurarei demonstrar aos desenvolvedores usuários de Delphi como instalar e configurar um ambiente profissional e produtivo, utilizando a IDE Lazarus e o compilador FreePascal. Também será demonstrado, como instalar e configurar diversos componentes, compatíveis com o Lazarus, além de relatos das principais diferenças e dificuldades encontradas para a adaptação de programadores Delphi.

Para que serve:

O Lazarus e FreePascal são projetos de código aberto de muito sucesso. O número de usuários e colaboradores desses projetos cresce vertiginosamente, o que faz deles uma ótima alternativa para todos os programadores de Object Pascal.

Em que situação o tema é útil:

Muitos desenvolvedores desejam compilar seus programas em outras plataformas como, por exemplo: Linux, WinCE, MacOS etc. Com o FreePascal isso é possível. O fato do Lazarus / FreePascal já nascerem multiplataforma, simplifica muito essa tarefa, mas mesmo assim vários aspectos devem ser observados, como veremos adiante. Outro fato, é que muitos desenvolvedores estão presos a IDEs antigas, como Delphi 5, 6 e 7, seja por questões financeiras ou pelas dificuldades técnicas na migração de uma nova IDE, nesses casos, a adoção do Lazarus como IDE é uma alternativa a ser considerada.

Lazarus e FreePascal:

Conheça a IDE Lazarus e o compilador FreePascal, e veja nesse artigo como criar um ambiente de trabalho que lhe permitirá criar aplicações profissionais, usando apenas ferramentas de código aberto. Além de permitir gerar código nativo em diferentes plataformas como Win32/64, Linux, WinCE, MacOS.

O que é o Lazarus?

Lazarus é um IDE. (Integrated Development Environment), ou ambiente de Desenvolvimento Integrado, para o compilador FreePascal. Ele é todo desenvolvido utilizando-se o próprio FreePascal. Muitos o consideram um "clone" do Delphi, pois as semelhanças com o Delphi 7 são imensas.

O que é o FreePascal?

O FreePascal ou FPC como também é conhecido, é um compilador 32/64 bits para a linguagem ObjectPascal, possuindo algum grau de compatibilidade com o ObjectPascal utilizado pela Borland nos compiladores do Delphi. Ele é capaz de gerar código nativo em Win32, Win64, WinCE, Linux, MacOS, FreeBSD, OS/2 e outros.

A equipe de desenvolvedores do FPC é independente da equipe do Lazarus, embora vários desenvolvedores atuem em ambos os projetos. O FPC foi iniciado em 1997 por Florian Klaempfl, com o intuito de suprir a lacuna deixada pela descontinuação dos compiladores Turbo Pascal da Borland.

Por que usar Lazarus se já existe o Delphi?

A escolha é antes de tudo pessoal. Nós como desenvolvedores não podemos ficar presos a rótulos, defendendo nossos IDEs a unhas e dentes, devemos ser produtivos. Se

seu IDE é produtivo e lhe proporciona tudo que você precisa, então continue sendo feliz com ele. Mas sempre analise os lançamentos de novas ferramentas que podem tornar o seu trabalho ainda mais produtivo, ou lhe permitam atingir novos objetivos.

O fato do Lazarus ser gratuito, pesou pouco na minha decisão, mas pode ser um ponto forte para muitos pequenos desenvolvedores que muitas vezes usam cópias antigas ou mesmo ilegais do Delphi. No meu caso, esbarrei em lacunas que o Delphi não foi capaz de preencher, como por exemplo, compilação para Linux e WinCE.

Fiz uso intensivo do Delphi 7 e Kylix 3 com a CLX. Desenvolvi sistemas inteiros usando a CLX, e minha decepção foi muito grande ao ver que ela foi simplesmente descontinuada. Nessa época eu já conhecia o Projeto Lazarus, mas sempre o vi com um certo ceticismo. Eu pensava comigo mesmo: "Devo realmente basear todo setor de desenvolvimento de minha empresa em uma ferramenta de Código aberto?" Assim como muitos, eu achava que estaria muito mais seguro com um produto comercial, e continuei usando o Delphi e Kylix. Após perceber que estava investindo em uma tecnologia abandonada (Kylix) decidi migrar em definitivo para o Lazarus. Para minha surpresa, após apenas 3 meses de estudo e conversão, os primeiros projetos já estavam sendo compilados, e continuei acreditando no Lazarus, até conseguir o que considero uma primeira vitória: a homologação de nosso PDV em TEF dedicado no mês de Abril de 2010.

Código Aberto

Os projetos Lazarus e FPC são desenvolvidos no regime de Código aberto, isso significa que eles não são propriedade de uma empresa ou pessoa específica, e também não dependem de único desenvolvedor. Assim como todos os projetos de código aberto, você mesmo pode estudar e contribuir com o código fonte do Lazarus, e é justamente esse fato que atrai cada vez mais desenvolvedores talentosos para o projeto.

Entretanto, você não tem autorização para gravar suas alterações diretamente no repositório de fontes do projeto, isso é feito pela equipe de desenvolvimento do Lazarus. Cabe a eles analisar as modificações propostas e decidir ou não utilizá-las. A equipe deve ser aberta a novas ideias, mas ela é soberana em sua decisão. Código aberto não é uma democracia, é uma "meritocracia", onde quem mais colabora, naturalmente ganha mais poder no projeto, sendo convidado pelos demais desenvolvedores a ser um membro da equipe com acesso direto aos fontes, após ter demonstrado suas boas qualidades e intenções.

A equipe de desenvolvedores do Lazarus é composta por programadores talentosos e autônomos, apaixonados pelo projeto. Eles dedicam horas de seu tempo livre para mantê-lo, sem receber remuneração alguma por isso, a não ser a satisfação pessoal, além de doações esporádicas ou tarefas premiadas.

A princípio, isso pode parecer estranho e surreal, trabalhar "de graça", mas cada um faz um pouco, e geralmente aquilo o que precisa, ou que considere interessante. As contribuições de cada indivíduo são somadas ao projeto original.

No meu caso, após o envio de várias contribuições, consegui satisfazer o meu ego, e com muito orgulho, passei a constar na lista de "Colaboradores" do projeto Lazarus.

Licença de uso

Todo projeto de código aberto deve ser regimentado por uma licença de uso, que define os termos do que pode ou não, ser feito com os fontes do projeto. A fim de assegurar que o Lazarus e FPC possam ser usados no desenvolvimento de aplicações comerciais, os desenvolvedores utilizaram um conjunto de licenças, entre elas (GPL2, LGPL e LGPL modificada).

O IDE e o compilador FPC usam a licença GPL, isso impede que o código fonte do Lazarus ou FPC (ou parte dele) sejam utilizados para criar um outro IDE ou compilador comercial, mas podem ser usados para criar outro projeto em código aberto.

As bibliotecas do Lazarus, que são compiladas de maneira estática no seu executável final, usam uma versão modificada da licença LGPL, garantindo assim, que você não seja obrigado a distribuir seus fontes. Para mais informações, leia o arquivo "COPYING.txt" que é instalado no diretório raiz do Lazarus.

Nota do DevMan

PDV são as siglas para o comum Ponto de Vendas. São aplicações conhecidas por realizarem a parte de "frente de caixa" ou "balcão". Você pode notar facilmente estes tipos de aplicações em caixas de supermercado, lojas de roupa e praticamente todos os comércios.

Nota do DevMan

TEF é a sigla de Transferência Eletrônica de Fundos. É a tecnologia que realiza a integração das vendas com as operadoras de cartões de crédito. Observe ao realizar uma compra em supermercado e passarmos o cartão de crédito na máquina de cartões, este aparelho conecta-se via internet e realiza uma transação entre a administradora e o cliente realizando operações de débito ou crédito por exemplo.

É seguro usar Código Aberto?

Certamente que SIM. Talvez sem saber, você já faça uso de vários projetos de código aberto, pois a grande maioria dos servidores da Internet é baseada em projetos de código aberto, como Linux, MySql, Apache, OpenSSL etc. Empresas conceituadas como a Google Inc., IBM, Novell usam largamente código aberto, além de produzirem diversos projetos.

Sempre fui usuário de vários projetos de código aberto, e hoje em dia, quase todas as soluções tecnológicas utilizadas em minha empresa usam projetos de código aberto, Linux, FireBird, MySQL, Lazarus e tantos outros. Aprendi a gostar tanto de projetos de código aberto, que passei de consumidor a fornecedor. Adotei o modelo de código aberto para a criação de componentes e rotinas para automação comercial, iniciando o projeto ACBr, em 2004. Para a criação desse projeto, utilizei o próprio fórum da ClubeDelphi (veja sessão Links ao final deste artigo).

Usar um projeto de código aberto pode significar independência, você não estará mais nas mãos de uma única empresa. E como os fontes estão sempre disponíveis, outro grupo de programadores sempre poderá dar continuidade ao projeto, ou criar um projeto paralelo. As licenças de

uso garantem os direitos e deveres de usuários e desenvolvedores.

Em alguns casos, pode ser um pouco mais trabalhoso usar um projeto de código aberto no lugar de um projeto comercial. Instalar o Lazarus no Windows é algo extremamente simples. Mas a falta de uma documentação mais completa e integrada como ocorre no Delphi, pode desanimar alguns desenvolvedores sem muita persistência.

O Lazarus está realmente pronto? Por que o número de versão ainda é 0.9.28?

Nenhum programa está completamente pronto, sempre há algo a ser desenvolvido ou melhorado. O fato do número de versão ainda não ter atingido 1.0 não significa que o projeto seja imaturo. O famoso projeto WINE demorou mais de 10 anos para atingir a versão 1.0.

Como não existem pressões comerciais para novos lançamentos, os números de versões perdem a importância de serem atualizados rapidamente, mas são atualizados de acordo com as modificações implementadas no projeto. Uma grande mudança no projeto modifica o número principal, novos lançamentos apenas atualizam o número secundário.

No caso do Lazarus, as versões que terminam em números pares são as "estáveis" e as ímpares são utilizadas em desenvolvimento. Portanto, no site oficial, você encontrará a versão 0.9.28 para download, enquanto os desenvolvedores trabalham na 0.9.29. Quando esta ficar pronta, será lançada como 0.9.30, e os desenvolvedores já iniciaram os trabalhos na 0.9.31.

O Lazarus é um projeto grandioso, e muita coisa ainda precisa ser implementada ou melhorada, e a equipe de desenvolvimento conta com a ajuda da comunidade para isso. Porém, em seu estágio atual, já é plenamente possível utilizá-lo para o desenvolvimento da maioria das aplicações comerciais.

Um pouco sobre a história do Lazarus

O Lazarus é a continuação de um projeto que havia sido encerrado, o projeto "Megido". Por isso o nome Lazarus foi escolhido (para os que não se lembram, Lazarus é o nome do personagem bíblico que foi ressuscitado por Jesus).

O projeto Megido, foi iniciado em 1998 e tinha como objetivo ser um Clone do Delphi. Os desenvolvedores do Megido utilizaram como base o código fonte de outro projeto, o Sybil, que era um Clone do Delphi para OS/2. Devido à falta de foco e interesse por um clone baseado nos fontes do Sybil o projeto Megido morreu em 1999.

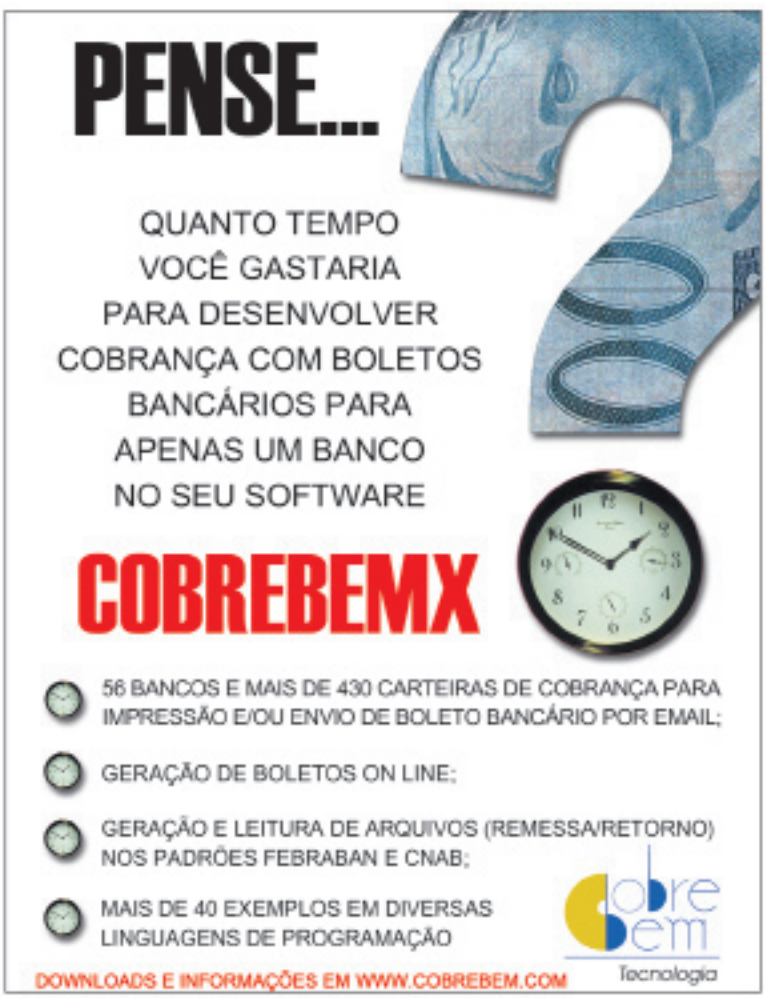
Alguns desenvolvedores, frustrados com os rumos do projeto Megido, tiveram a iniciativa de fundar um novo projeto, nascia assim o Lazarus. Cliff Baeseman, Shane Miller e Michael A. Hess, iniciaram o Lazarus em fevereiro de 1999. Sendo que desses três, apenas Michael A. Hess continua no projeto até hoje.

Além de Michael, os membros mais antigos do time são Marc Weustink e Mattias Gaertner. Ambos são os maiores contribuidores para o núcleo do que é o Lazarus hoje. Atualmente vários desenvolvedores, de diversos países, têm papel importante no desenvolvimento do Lazarus, entre eles podemos citar: Vincent Snijders, Micha Nelissen, Paul Ishenin, Michael Van Canneyt, Jesus Reyes, e inúmeros outros.

O Brasileiro Felipe Monteiro de Carvalho, aluno de engenharia da USP, ingressou na equipe de desenvolvedores em 2005, ele atualmente escreve um livro sobre o Lazarus e mantém o suporte do Lazarus ao WinCE.

Multiplataforma

O Lazarus e FPC são multiplataforma, ou seja, rodam em diversos sistemas operacionais. O lema do Lazarus é "Write once, compile anywhere", traduzindo: "escreva uma vez, e compile em qualquer lugar". Porém na prática, isso não é tão simples assim, a não ser que o programa seja um simples, "Olá Mundo". Mas sim, é possível recompilar o seu projeto Win32 em MacOS e Linux, desde que você não



utilize funções de baixo nível específicas do sistema operacional.

O que os desenvolvedores do Lazarus esperam dizer com essa frase, é que o Lazarus / FPC é capaz de gerar código nativo, para todas as plataformas suportadas, diferente de Java ou .NET, onde o código compilado não é nativo, e depende de uma máquina virtual compatível com Sistema Operacional. Código nativo tem grande velocidade, e independe de um módulo Runtime para a sua execução.

Realizando o download do Lazarus

Depois de tanta teoria e de conhecermos bem a nossa ferramenta vamos ao que realmente interessa, vamos baixar e instalar o Lazarus. Como citado anteriormente, a versão estável do Lazarus é a 0.9.28, e o link de download pode ser encontrado na sessão Links deste artigo.

Não efetue o download ainda. Não recomendo usar essa versão, ao invés disso, vamos baixar uma versão de Snapshot (compilação noturna), pois importantes modificações e correções de bugs já foram introduzidas no Lazarus e FPC desde que a 0.9.28 foi lançada, o que tornam essa versão obsoleta (em minha opinião). Possivelmente, até a publicação deste artigo, a versão 0.9.30 seja finalmente lançada, o que nesse caso, representa ser uma melhor opção para a instalação.

O Link para download dos "Snapshots" também pode ser encontrado na página principal do projeto, logo abaixo do link "Downloads", você encontrará o link "Daily Snapshots".

Na página dos Snapshots podemos ler um aviso que diz: "CUIDADO: Estes Snapshots são gerados automaticamente e não são testados. A única coisa que podemos dizer é que o compilador encontrou os fontes bons o suficiente para compilar." Não se preocupe com esse aviso alarmante, os Snapshots são uma ótima maneira de termos versões intermediárias do Lazarus entre o lançamento das versões estáveis, sem precisar recorrer a ferramentas mais complexas, como por exemplo o SVN.

Os Snapshots são instaladores completos do Lazarus, gerados diariamente pelo servidor, através da compilação dos fontes atuais do repositório (SVN). Isso significa que em algumas raras ocasiões, você pode baixar uma versão que tenha algum bug, que acidentalmente foi introduzido como efeito colateral por alguma modificação em andamento. Isso é raro, pois os desenvolvedores do Lazarus procuram ter muito cuidado com o código fonte que enviam para o repositório de fontes. Se você notar que a versão instalada possui um grave bug, basta baixar um novo instalador do Snapshot nos dias seguintes, e proceder com a instalação sobre a versão atual.

Na página dos Snapshots, existem várias opções para download, procure pelo item descrito como: "Lazarus + FPC 2.4.3 win32". Mesmo que seu Windows seja 64 bits, prefira a versão Win32, afinal seu binário deverá rodar em máquinas com várias versões do Windows. Se você também deseja compilar para WinCE, baixe o complemento desta versão descrito como: "Lazarus + FPC 2.4.3 win32 -> arm wince".

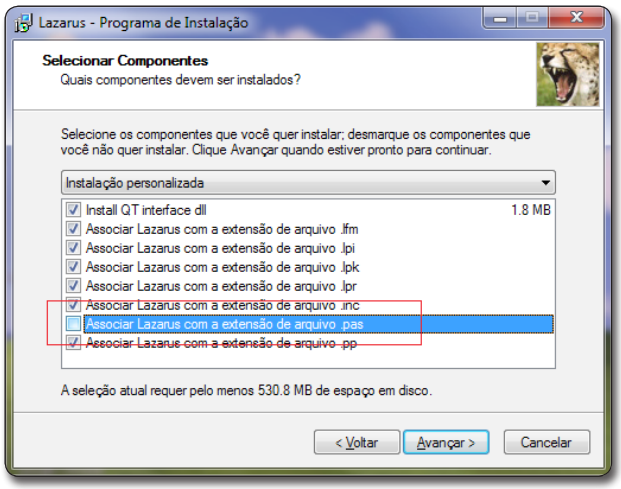


Figura 1. Evitando conflitos de extensão .pas com o Delphi

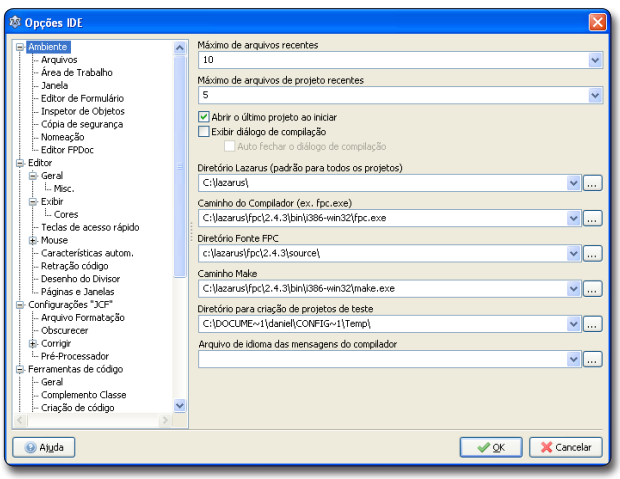


Figura 2. Configurando a integração com o compilador FPC

Instalando o Lazarus

Instalar o Lazarus no Windows é um processo extremamente simples. Tratando-se basicamente do tradicional "Next, Next e Finish". Se você tem receio de que a instalação do Lazarus possa comprometer o seu ambiente de trabalho com o Delphi, faça uso de uma máquina Virtual como, por exemplo, o Virtual Box ou VMWare. Porém, o único conflito que o Lazarus pode causar com o Delphi é em relação ao registro das extensões de arquivo *.pas, mas isso pode ser facilmente contornado durante sua instalação, desmarcando a opção correspondente, conforme mostra a **Figura 1**.

Após clicar em "Avançar" em mais algumas telas, o instalador estará apto a proceder com a cópia dos arquivos. Por padrão, o Lazarus é instalado no diretório "c:\lazarus". Isso é bom, pois evita problemas de permissão de escrita, principalmente nos novos Windows Vista ou 7. Se tudo ocorrer corretamente, após a instalação você já verá no seu Desktop o Ícone do Lazarus, cujo símbolo é um Guepardo.

O compilador Free Pascal (FPC) é instalado no diretório "c:\lazarus\fpc" e o instalador deve configurar corretamente os arquivos de configuração para a IDE encontrá-lo, mas se isso não ocorrer, você pode configurar o ambiente de compilação em: Environment> Options, conforme demonstra a **Figura 2**.

Conhecendo a IDE do Lazarus

Para quem usa ou já usou Delphi 7 ou anterior, o IDE do Lazarus não tem segredos. A ideia dos desenvolvedores do Lazarus

era criar um Clone do Delphi, e temos que admitir que eles tiveram sucesso nesse objetivo. Na **Figura 3** temos uma visão do IDE já instalado.

A semelhança com o Delphi 7 não para apenas no visual. Praticamente todas as teclas de atalho e botões da barra de ferramentas têm o mesmo funcionamento do Delphi, ou seja:
F9 - Compila e executa;
F11 - Traz o foco para o Object Inspector;
F12 - Alterna entre o Editor de Código e o Desenhador de Formulário;
Shift-F12 - Mostra a lista de Formulários.

Olá Mundo

Para experimentar o IDE vamos fazer um simples projeto de "Olá Mundo". Observe que assim como o Delphi 7, o Lazarus quando iniciado já exibe um projeto vazio com um formulário, caso isso não ocorra, você pode iniciar um projeto a qualquer momento escolhendo as opções: "Project>New Project>Application".

Insira um botão no Formulário, da mesma maneira que você está acostumado a fazer no Delphi, ou seja, clique no componente TButton da aba Standard da paleta de componentes, e depois clique no formulário. Observe que você pode redimensionar e arrastar o componente da mesma maneira como é feito no Delphi.

Dê um clique-duplo no botão para invocarmos a criação do seu método padrão (OnClick). Assim como no Delphi o IDE do Lazarus criará a declaração do método, e posicionará o editor no método recém-criado.

Vamos escrever um simples comando no evento, para completar o nosso programa "Olá Mundo", mas nesse meio tempo, experimente o Code Completion. Escreva: "showm" e tecle "Ctrl+Space". Veja na **Figura 4**, que o IDE, irá mostrar todos os métodos e identificadores que iniciam com o texto digitado.

Complete a instrução escolhendo a primeira versão da procedure ShowMessage e digite o texto:

"ShowMessage('Olá Mundo');"

Tudo pronto, basta pressionar F9 para rodar o seu primeiro projeto em Lazarus. Veja o resultado na **Figura 5**.

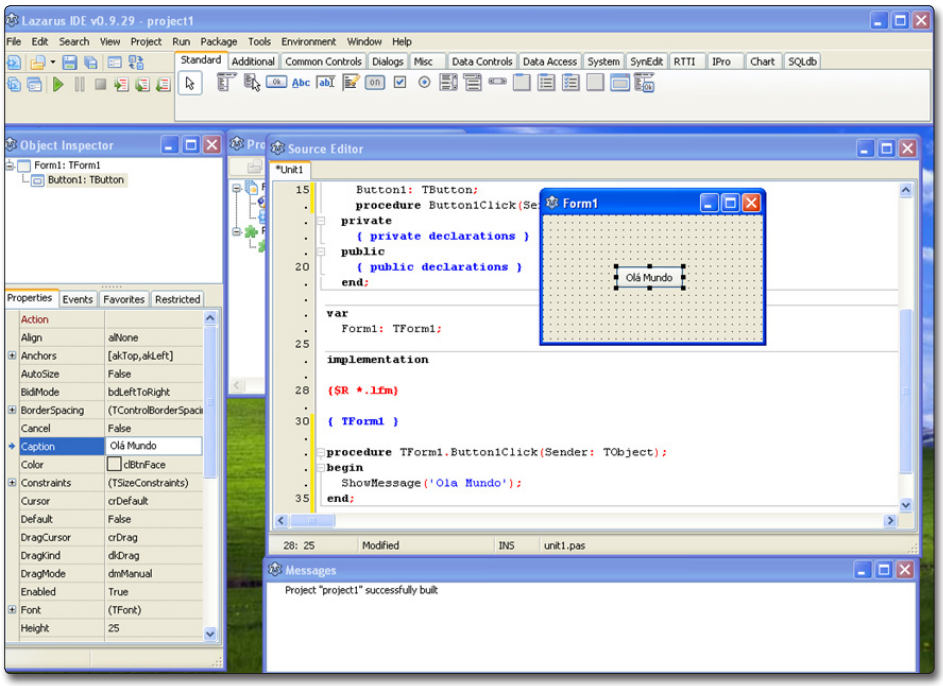


Figura 3. Visão geral do IDE do Lazarus

IDE estilo RAD Studio

O IDE do Lazarus pode parecer antiquado para os usuários de Delphi 2005 ou superior. O estilo de "janelas soltas" foi muito popular até o Delphi 7, mas hoje em dia, os IDEs modernos não mais utilizam esse visual. No entanto, após configurar as janelas em suas devidas posições e dimensões, a produtividade é muito boa, principalmente para desenhar formulários grandes.

Os desenvolvedores do Lazarus já estão trabalhando num complexo mecanismo de acoplamento, que fará parte da base dos componentes do Lazarus. Com ele, será possível acoplar as janelas do IDE umas nas outras. Veja na **Figura 6** uma prévia de como ficará o IDE usando o mecanismo de acoplamento. Essa nova característica é aguardada para a versão 0.9.30.

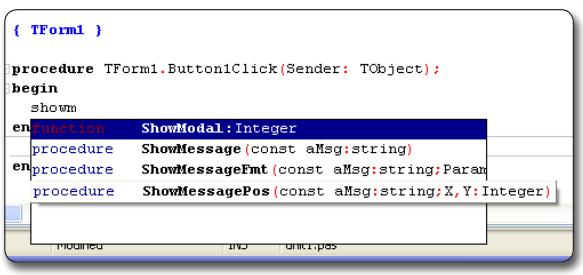


Figura 4. Code Completion em ação

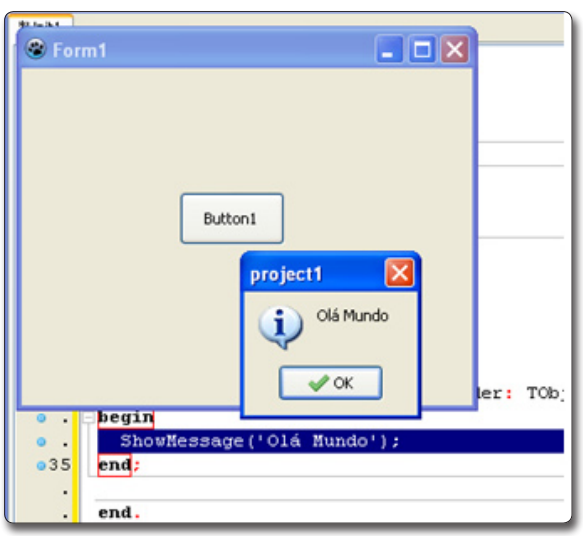


Figura 5. Executando um "Olá Mundo"

Configurações úteis para o IDE

Agora que já nos sentimos "em casa" devido à imensa semelhança com o Delphi, vamos efetuar pequenos ajustes nas opções do ambiente do IDE para aumentarmos a produtividade. O Lazarus tem uma infinidade de parâmetros de configuração para o IDE. A seguir relatarei apenas algumas

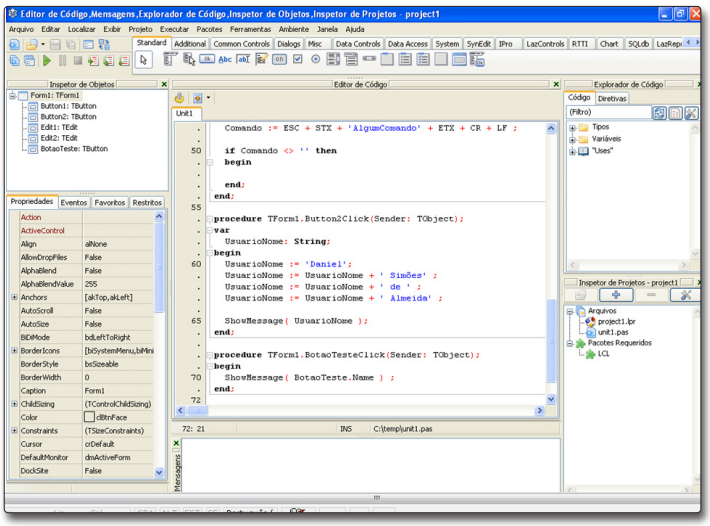


Figura 6. Prévia do IDE com a função de acoplamento

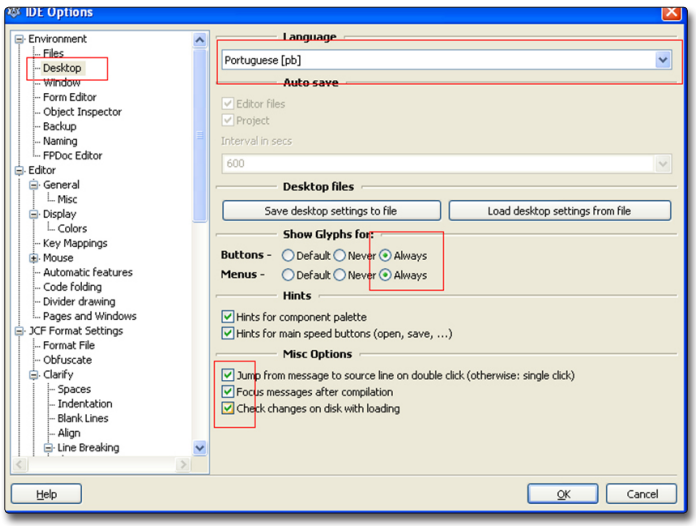


Figura 7. Opções do IDE

configurações que acho serem importantes para quem está acostumado com o Delphi. No entanto, recomendo a todos pesquisar ao máximo as opções existentes, pois analisando as opções de configuração do ambiente, é possível conhecer algumas características únicas do Lazarus, como por exemplo, a integração com o formatador de código JEDI e a integração com o gerador de documentação FPDoc.

Configurando o IDE para Português

O Brasil é um dos países com o maior número de adeptos do Lazarus, e por isso, vários colaboradores se prontificaram em traduzir os arquivos de língua do Lazarus para o Português. Basta configurar a língua do IDE em: Environment>Options>Desktop>Language. Após a configuração, reinicie o Lazarus para visualizar o IDE em Português. Observe as configurações na **Figura 7**.

Exibindo ícones nos botões e menu

Na mesma tela da **Figura 7**, aproveite e configure para o Lazarus exibir os ícones dos botões e menus, marque "Always" em ambas as opções de "Show Glyphs for". Isso dá uma estética melhor às janelas de mensagem e no IDE.

Opções diversas

Selecione também na **Figura 7** todas as opções de "Misc Options" (ou Opções diversas se você já está em Português). A primeira opção impede o Lazarus de desviar o foco para o editor quando você dá um clique simples na Janela de mensagens (isso é irritante). Após isso, quando você realmente quiser pular da janela de mensagens para a linha indicada no editor, use um clique duplo na janela de mensagens.

A terceira opção fará o IDE monitorar se os arquivos que estão abertos no editor sofreram alterações através de outros programas, e se for o caso, sugerir que você recarregue o arquivo.

Desabilitando alguns alertas do Debug

Eu pessoalmente não gosto de muitas interrupções quando estou executando o programa dentro do IDE. O Lazarus vem configurado para alertar quando o programa foi encerrado, o que a meu ver, é desnecessário. Para desabilitar isso, basta clicar no botão apropriado (**Figura 8**).

Outra opção que costumo desabilitar são as notificações de exceções. Como trabalho muito com o a criação de exceções em tempo de execução, para controlar o fluxo do programa, essa opção me incomoda. Para isso acesse a opção Environment>Options>Debugger>Language Exceptions, conforme mostra a **Figura 9.**

Habilitando a remoção automática de métodos vazios

No Delphi estamos acostumados a deixar a cargo do IDE remover os métodos vazios. Isso ocorre, por exemplo, quando você clica em um evento, e o IDE cria automaticamente o cabeçalho do método para você, mas por um motivo qualquer, você decide que não vai utilizá-lo. Nesse caso, o Lazarus por padrão não apaga o evento vazio, mas podemos configurá-lo para que isso ocorra. Para isso acesse a opção Environment>Options>Editor>Automatic Features (**Figura 10**).

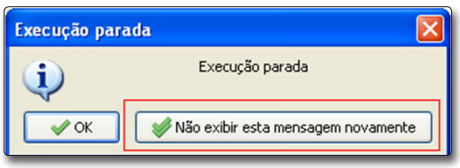


Figura 8. Mensagem de fim de execução

Nota do DevMan

Um widgetset é um conjunto de controles ou componentes visuais, que compõem uma interface gráfica com o usuário (GUI). Todas as aplicações visuais são compostas de vários Widgets, como por exemplo: o Edit, ComboBox, RadioButton, Grids etc.

Arquivos de backup

Os Backups são muito importantes. O Lazarus oferece uma grande variação de métodos para criá-los. Eles serão criados sempre que você gravar os seus fontes (Ctrl+S). Eu achei prático criar os arquivos de Backup em uma subpasta separada dos fontes, para isso configure a criação dos arquivos de backup conforme a **Figura 11**. A extensão do arquivo original é preservada. Exemplo: Unit1.pas será copiado para backup\Unit1.pas.bak

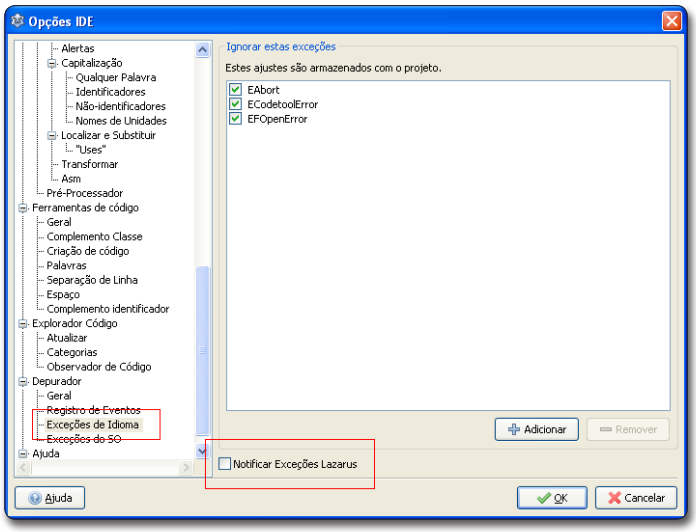


Figura 9. Desabilitando as notificações de exceções

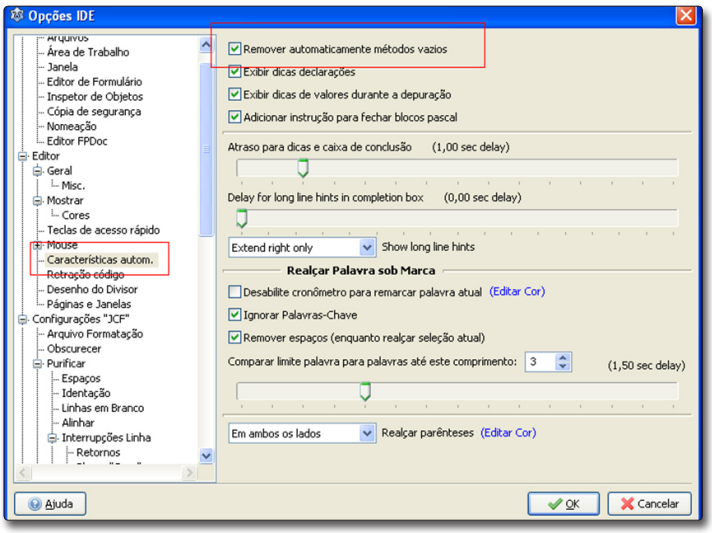


Figura 10. Habilitando a remoção automática de métodos

Sem suporte a MDI

Acredito que esse seja um dos pontos que mais incomodam o desenvolvedor Delphi. O Lazarus não tem, e provavelmente nunca terá, suporte a MDI (Interface de Múltiplos Documentos). A explicação é que não é possível implementar MDI de forma nativa, em todos os Widgetsets suportados pelo Lazarus. Poderíamos ter MDI na Win32 e QT, mas não na GTK2. Os desenvolvedores do Lazarus também alegam que a MDI "caiu de moda", e que mesmo a Microsoft vem recomendando evitar o uso da MDI. A discussão sobre as vantagens e desvantagens da MDI não estão no escopo deste artigo.

Dificuldade na depuração dos fontes (Debug)

Essa sem dúvida é atualmente a maior deficiência do Lazarus. O depurador usado pelo Lazarus atualmente é o GDB, que faz parte do projeto GNU. Você conseguirá definir Breakpoints da mesma maneira como é feito no IDE do Delphi (F5). As teclas de atalho também são as mesmas, F7 (Step Into), F8 (Step Over), F9 (Run) e etc. Além das teclas tradicionais, use Ctrl+Alt+B para manipular a lista de todos os Breakpoints.

O problema maior é no caso do Object Inspector. Você conseguirá inspecionar o conteúdo de uma variável local, posicionando o mouse sobre ela e aguardando alguns segundos. Mas no caso de propriedades de Objetos ou Classes, o depurador apenas conseguirá exibi-la, se a propriedade for mapeada diretamente para uma variável interna da Classe, em sua cláusula "read", e não para uma função. Isso incomoda bastante, e torna a depuração menos eficiente.

Uma alternativa para inspecionar valores de propriedades é criar arquivos de LOG, e gravar em cada linha do arquivo, o valor das propriedades em tempo de execução. Outra alternativa seria modificar os fontes para atribuir as propriedades que você deseja inspecionar para variáveis locais (**Figura 12**). Ou ainda, usar um ShowMessage para exibir o valor da propriedade.

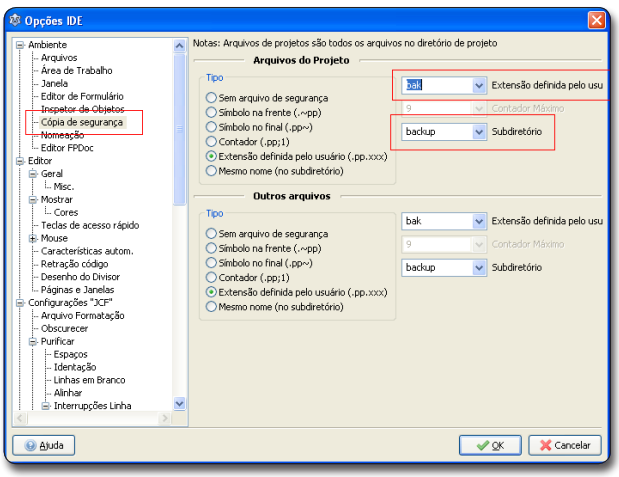


Figura 11. Configurando a criação de arquivos de backup

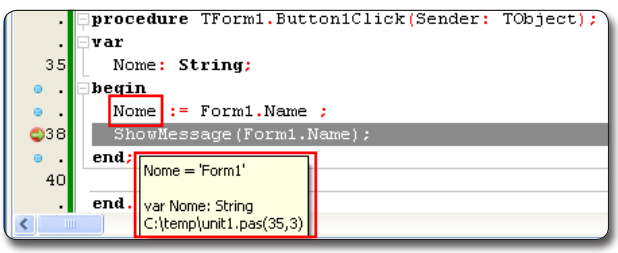


Figura 12. Inspecionando variáveis internas

Suporte a caracteres Unicode

Desde a versão 0.9.26, o Lazarus já suporta nativamente caracteres com a codificação Unicode. No Delphi, o Unicode foi introduzido a partir da versão 2009. Se você nunca escreveu Software multiplataforma ou de uso internacional, então provavelmente nunca precisou (ou desejou usar) o Unicode. Caso contrário já deve ter percebido como é difícil distribuir um Software que será executado em diversas plataformas, ou em diversas línguas.

O Unicode é um formato de página de código de caracteres que permite representar toda e qualquer espécie de carácter, de todas as línguas, existentes e imagináveis, mesmo em diferentes plataformas.

No Brasil, a página de código usada por padrão no Windows, é a CP1252. Se o seu programa foi compilado por um Delphi

que não suporta Unicode, ele provavelmente terá problemas para rodar em Windows de outras linguagens, que não usem a página de código CP1252.

No Lazarus obrigatoriamente precisamos adotar o Unicode. Isso significa que todas as units e formulários do projeto devem estar codificadas em Unicode. Se você está criando um novo projeto no Lazarus, não se preocupe, todas novas units e dormulários já usaram o Unicode (UTF-8). Porém, se você está migrando uma aplicação do Delphi para Lazarus, provavelmente essas units e formulários estão usando a página de código do Windows (CP1252), e nesse caso você precisará converter esses arquivos para que os caracteres acentuados apresentem-se da forma correta.

Você pode verificar a codificação e modificá-la usando o próprio editor de textos do Lazarus. Clique com botão direito do Mouse no arquivo aberto no editor, e escolha a opção "Configurações Arquivo>Codificação" e observe na lista apresentada qual o tipo atual, para converter, basta escolher um novo tipo. Observe o exemplo na **Figura 13**.

Lembre-se que os formulários (*.dfm) também precisam ser convertidos para UTF8. Para isso, primeiro você deve converter o dfm para texto, usando o IDE do Delphi, em seguida, abrir o arquivo .dfm no editor do Lazarus e verificar a codificação da mesma maneira que fizemos com o .pas. Modifique-o para UTF-8 se necessário, como mostra o exemplo da **Figura 14**.

Se o seu projeto requer o uso de banco de dados, ele também deve ser configurado para usar Unicode, caso contrário você poderá ter problemas quando recuperar as informações do mesmo.

Também é importante mencionar que com a chegada do Unicode, temos que ter um cuidado extra com a definição de tipos de variáveis. Nas versões antigas do Delphi e Lazarus, o tipo String era mapeado para o tipo AnsiString, já nas novas versões que suportam Unicode, o tipo String passou a ser mapeado para o Tipo UnicodeString.

O AnsiString comporta-se como um String Binário, podendo armazenar qualquer tipo de informação, já o UnicodeString, deve apenas conter Strings que sejam codificadas em Unicode. Portanto, se você tem algum código que armazene caracteres de controle em Strings, ele precisará ser revisto para que as declarações de variáveis do tipo String sejam substituídas por AnsiString. O exemplo da **Listagem 1** mostra como devem ser declaradas as variáveis que poderão conter caracteres de controle. Essa mudança é completamente funcional com as versões antigas do Delphi.

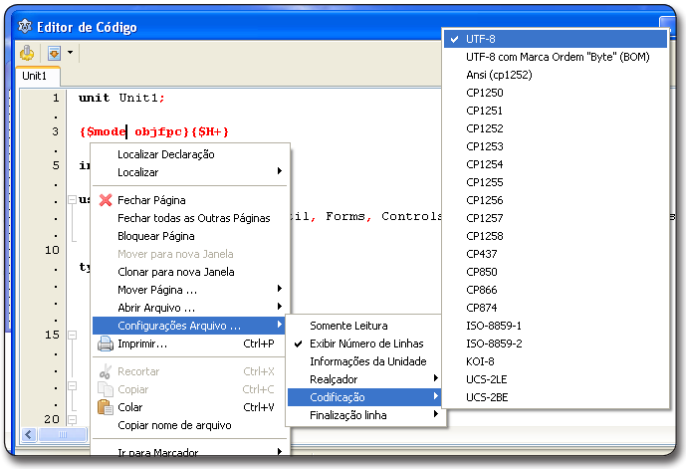


Figura 13. Verificando a codificação do arquivo fonte

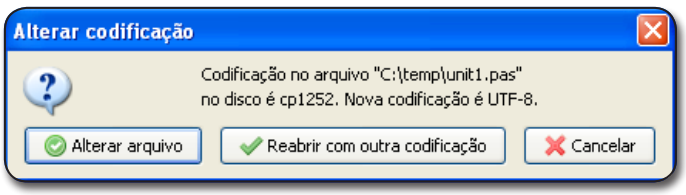


Figura 14. Confirmação para mudança de codificação

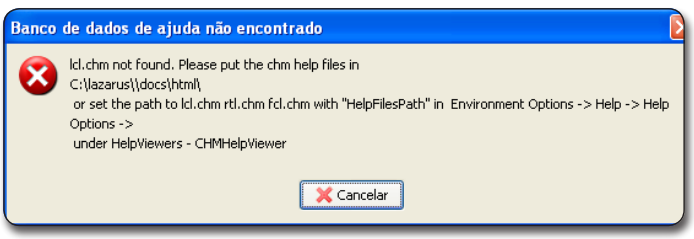


Figura 15. Mensagem de erro informando a falta dos arquivos CHM

Configurando os arquivos de ajuda

Atualmente o Lazarus usa arquivos de ajuda no formato CHM. O suporte a esse tipo de arquivo é relativamente novo no Lazarus / FPC. Isso porque os desenvolvedores do Lazarus queriam uma solução multiplataforma e a manipulação de arquivos CHM no Windows, geralmente depende do uso de APIs do Windows e, portanto, não funcionaria em outras plataformas. Os desenvolvedores do Lazarus / FPC decidiram então escrever rotinas de suporte ao formato CHM, totalmente em Pascal, e isso foi feito. Isso possibilitou a criação de um "engine" para leitura e geração de arquivos CHM, e permite ao usuário do Lazarus / FPC usá-lo nativamente em múltiplas plataformas.

O IDE do Lazarus tem seu próprio visualizador de CHM (todo escrito em Lazarus / FPC), porém, ele não é previamente compilado pelo instalador do Lazarus. Para invocarmos a sua compilação, basta tentar usar a ajuda pela primeira vez. Exemplo: Posicione o cursor no editor, sobre a palavra TForm, e pressione F1. Observe que o IDE irá invocar um Script para compilar o visualizador de CHM, porém como os arquivos CHM ainda não foram instalados, ela retornará a mensagem de erro da **Figura 15**.

Se você baixou a versão de Snapshot do Lazarus como sugerido por este artigo, então não possui os arquivos de ajuda. Na

Listagem 1. Declaração correta de Strings binários

```
procedure TForm1.Button1Click(Sender : Object);
const
  STX = #2;
  ETX = #3
  CR  = #13;
  LF  = #10;
  ESC = #27;
var
  Comando : AnsiString;
begin
  Comando := ESC + STX + 'AlgumComando'
    + ETX + CR + LF;
end
```

Nota

Antes do suporte a CHM, o Lazarus utilizava uma ajuda on-line, fazendo chamadas ao Browser, e apontando para as páginas de documentação do projeto. Ainda hoje, essa ajuda pode ser utilizada, através das opções do menu: "Ajuda>Ajuda OnLine", porém ela não é sensível ao contexto como ocorre no caso do CHM, cabe ao usuário navegar nas páginas e encontrar o que precisa.

sessão Links você encontra o endereço de onde eles podem ser baixados. Procure pelo arquivo contendo o texto CHM, é mais atualizado. Provavelmente o arquivo estará compactado no formato BZ2 (muito comum no Linux), para abri-lo no Windows, use um descompactador que suporte BZ2, como por exemplo, o 7-Zip. Descompacte os arquivos CHM no diretório: "C:\lazarus\docs\html", após isso, já teremos a ajuda integrada ao IDE funcionando, conforme a **Figura 16**.

Infelizmente os arquivos de ajuda do Lazarus são bem inferiores aos do Delphi. Ainda há muito que escrever nos arquivo de ajuda do Lazarus, e essa tarefa pode ser complementada pela comunidade de usuários. A documentação de vários objetos está incompleta, e são poucos os exemplos existentes. A boa notícia é que praticamente todas as informações do Delphi também se aplicam ao Lazarus / FPC, de maneira que você pode continuar usando os manuais de ajuda do Delphi.

Code Colapse de blocos

O editor do Lazarus automaticamente detecta o início e fim de blocos de código, como por exemplo, procedures, comandos condicionais, loops entre outros recursos, inserindo marcações à esquerda do editor, que permitem fechar e expandir esses blocos, a fim de facilitar a leitura de rotinas extensas (**Figura 17**).

Favoritos no Inspetor de Objetos

O Inspetor de Objetos possui uma aba chamada "Favoritos" onde aparecem as principais propriedades daquele determinado objeto, ou as que você definiu como favoritas. Para definir uma propriedade como favorita, clique com o botão direito do mouse sobre a mesma, e selecione "Adicionar aos Favoritos". A aba favoritos quando corretamente usada, proporciona uma maneira rápida de achar as propriedades que você usa com mais frequência. Observe o exemplo na **Figura 18.**

Auto identação e Code Complete

O editor é capaz de analisar a identação dos seus fontes, e enquanto você escreve, ele posicionará o cursor de acordo com a identação usada. Observe na **Figura 19**, que após escrever a palavra "begin" abaixo do "if", e teclar "Enter", o editor já completa o bloco inserindo o texto "end;" no mesmo alinhamento do "begin" (recurso de Code Complete), e posiciona o cursor dentro do bloco, com identação correta.

Auto declaração de variáveis

Este é o meu favorito. O editor pode declarar as variáveis para você, desde que a sua expressão de atribuição esteja bem definida. Exemplo: supondo que estejamos escrevendo a atribuição de uma variável do tipo String, chamada "Nome" (**Figura 20**).

Note que até o momento, a variável Nome ainda não foi declarada em "var", aliás, nem ao menos existe a seção "var" na procedure. Agora vamos executar a mágica. Posicione o cursor sobre a variável que deseja declarar e pressione Ctrl+Shift+C. A declaração será feita pelo editor, observe que o editor cuida até mesmo da identação correta, conforme a **Figura 21**.

Substituição em bloco (Syncron-Edit)

Esse recurso é bastante interessante. Ele permite modificar em um bloco de código previamente selecionado, qualquer ocorrência de uma palavra que se repita nesse bloco. Acho que a melhor maneira de explicá-lo é demonstrando-o. Suponha que eu tenha o código da **Figura 22** e desejo alterar todas as

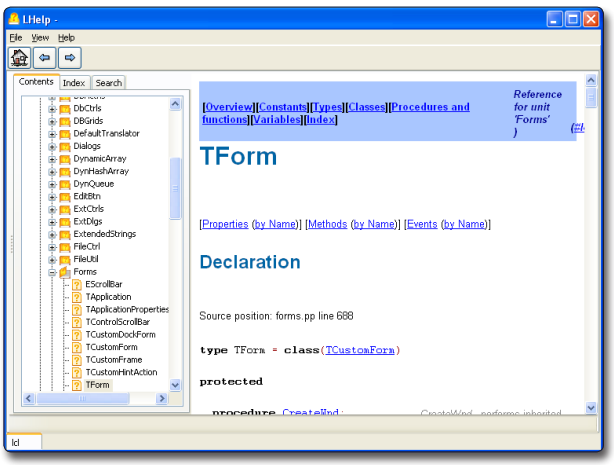


Figura 16. Arquivo de ajuda no Lazarus

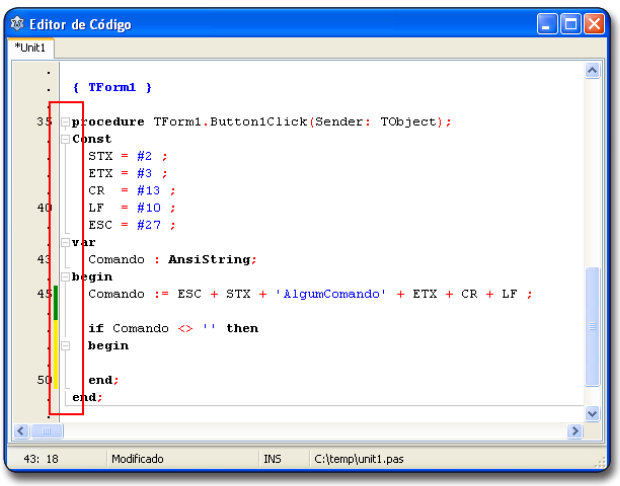


Figura 17. Destaque das marcações do Colapse editor

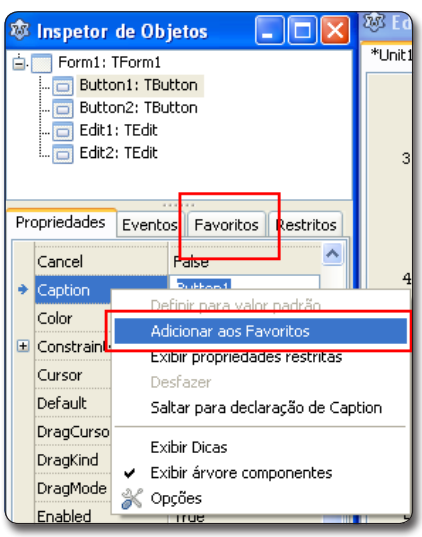


Figura 18. Aba favoritos do inspetor de objetos

ocorrências da variável "Nome" para "Usuario". Isso é um trabalho para o Syncron-Edit. Selecione todo o bloco, e depois clique

```
  if Comando <> '' then
  begin
    |
  end;
end;
```

Figura 19. Exemplo de Code Complete

```
55 procedure TForm1.Button2Click(Sender: TObject);
56 begin
     Nome := 'Daniel';
   end;
```

Figura 20. Escrevendo uma variável não declarada

```
55 procedure TForm1.Button2Click(Sender: TObject);
   var
     Nome: String;
   begin
58   Nome := 'Daniel';
   end;
```

Figura 21. Declaração automática da variável

```
55 procedure TForm1.Button2Click(Sender: TObject);
   var
     Nome: String;
   begin
58   Nome := 'Daniel';
     Nome := Nome + ' Simões' ;
60   Nome := Nome + ' de ' ;
     Nome := Nome + ' Almeida' ;

     ShowMessage( Nome );
   end;
```

Figura 22. Exemplo de código

```
55 procedure TForm1.Button2Click(Sender: TObject);
   var
     Nome: String;
   begin
     Nome := 'Daniel';
     Nome := Nome + ' Simões' ;
60   Nome := Nome + ' de ' ;
     Nome := Nome + ' Almeida' ;

     ShowMessage( Nome );
64 end;
```

Figura 23. Clicando no botão do Syncron-Edit

```
55 procedure TForm1.Button2Click(Sender: TObject);
   var
     UsuarioNome: String;
   begin
     UsuarioNome := 'Daniel';
59   UsuarioNome := UsuarioNome + ' Simões' ;
60   UsuarioNome := UsuarioNome + ' de ' ;
     UsuarioNome := UsuarioNome + ' Almeida' ;

     ShowMessage( UsuarioNome );
   end;
```

Figura 24. Modo Syncron-Edit em ação

no botão que aparece no canto esquerdo do editor, no fim do bloco (**Figura 23**). O editor entrará no modo Syncron-Edit. Agora basta modificar qualquer ocorrência da palavra "Nome" para "Usuario". Para sair do modo Syncron-Edit basta clicar novamente no botão, que agora está no topo do Bloco (**Figura 24**).

Renomeando componentes

Quando você modifica a propriedade "Name" de um componente no Inspetor de Objetos, o IDE cuida de renomear todas as ocorrências deste componente no seu código fonte. Isso é muito útil para evitar erros de compilação.

Exemplo: Suponha que você possua um Form com um botão chamado "Button3", e já tenha escrito algum código que faça referência a esse componente em sua Unit. Após um tempo, você decidiu modificar a propriedade "Name" deste botão para algo mais significativo, como por exemplo: "btAtivar". O IDE ajustará todo código fonte desta Unit substituindo as referências de "Button3" para "btAtivar".

Mudando o componente de classe

Em algumas situações desejamos trocar um componente por outro semelhante. Porém quando apagamos o componente do editor de Formulário, perdemos todas as configurações efetuadas para ele. O Lazarus tem um recurso muito interessante, que permite mudar a classe "Pai" do componente. Exemplo: Suponha que você possui um formulário com um botão do tipo TButton, e deseja mudá-lo para o tipo TBitBtn, para poder inserir uma imagem. Clique com o botão direito do mouse no botão e escolha a opção "Alterar Classe", como mostra a **Figura 25.**

Após isso, o IDE apresentará o diálogo para você informar a nova classe do componente. Escolha o novo Tipo (TBitBtn) e clique em "OK" (**Figura 26**).

Em alguns casos, a nova classe não possuirá total compatibilidade de propriedades

Nota

Caso seja necessário, o IDE adicionará a dependência de novas Units para atender a esse novo tipo de classe, como mostra a **Figura 27**.

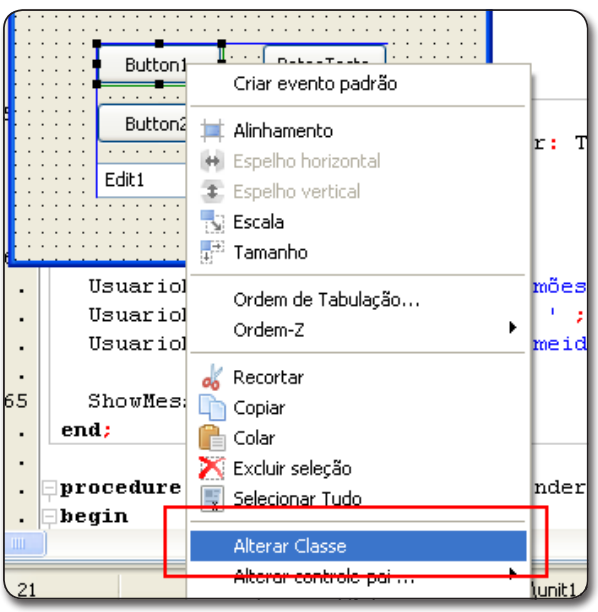


Figura 25. Alterando a classe de um componente

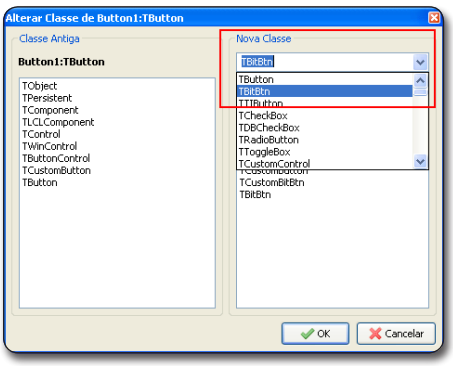


Figura 26. Selecionando a classe a ser alterada

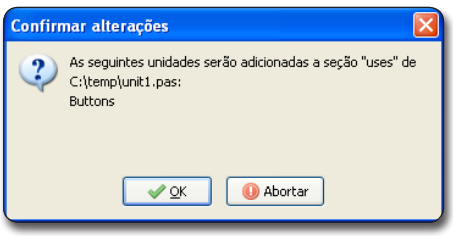


Figura 27. Confirmação de adição de Unit à sessão uses

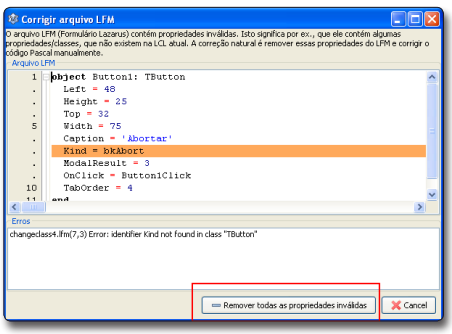


Figura 28. Aviso de perda de propriedades

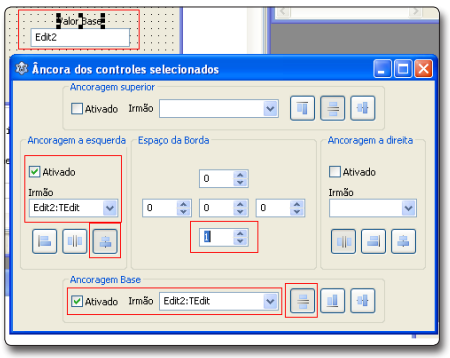


Figura 29. "Ancorando" componentes

des com a anterior. Quando isso ocorrer, o IDE apresentará um diálogo, informando as propriedades que serão perdidas. Clique em "Remover todas propriedades inválidas". Isso ocorrerá, por exemplo, quando mudarmos um TBitBtn para TButton, conforme a **Figura 28**.

O editor de âncoras

Quando você escreve programas multiplataforma, é fundamental usar bem os recursos de âncoras. Isso porque os componentes podem variar de tamanho, nas diversas plataformas suportadas, e as fontes nativas também mudam de tipo e tamanho, o que altera muito o desenho do formulário final. Com as âncoras, o seu programa saberá se ajustar ao novo ambiente.

Para exibir o editor de âncoras, clique no menu principal, em "Exibir>Editor de Âncora". Você poderá não somente "ancorar" o componente à direita, esquerda, topo e base, como também, "ancorar" um componente ao outro. Observe a **Figura 29**.

O componente TLabel1 (Caption "Valor Base") está "ancorado" ao componente TEdit2. Ele é "ancorado" pela esquerda, mas usando um alinhamento pelo centro. E "ancorado" pela base, ao Topo do TEdit2, usando um separador de borda de 1 pixel. Após essa configuração, onde o componente TEdit2 for desenhado, o TLabel1 irá segui-lo e se ajustar de acordo com as configurações de ancoragem.

Conclusão

Nesta parte do artigo conhecemos o IDE do Lazarus, seus recursos, aprendemos a configurar o ambiente, conhecemos todas as facilidades e dicas para se desenvolver utilizando este ambiente. Vimos também a semelhança e facilidade de reaproveitar nosso conhecimento em Delphi ao utilizar o Lazarus. Na próxima parte deste tutorial, veremos como instalar componentes no Lazarus e conheceremos um pouco mais sobre este projeto.

Daniel Simões Almeida
daniel@djsystem.com.br
Sócio/proprietário da empresa DJSystem Consultoria e Automação Comercial. Idealizador e principal mantenedor do Projeto ACBr (Automação Comercial Brasil). Graduado em Processamento de Dados e pós-graduado em Análise de Sistemas pela universidade UNIMEP (Universidade Metodista de Piracicaba). É responsável pelo setor de tecnologia e desenvolvimento da DJSystem, programando em ObjectPascal, xBase e PHP, nas plataformas Windows e Linux.

Fórum DevMedia onde o projeto ACBr teve início
http://forum.devmedia.com.br/viewtopic.php?t=42909

Download do Lazarus
http://www.lazarus.freepascal.org/

Arquivos CHM de ajuda
ftp://ftp.freepascal.org/pub/lazarus/snapshots/

Dê seu feedback sobre esta edição!

A .NET Magazine tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Criando um grand

Aprenda como criar um software apreciado

Se existe alguma verdade absoluta sobre Softwares é que eles sempre mudam. Não importa o quão perfeito eles estão, sempre muda um requisito aqui e aparece outro ali. Por essa e por outras que o sistema precisa ser o mais flexível possível, pois seu chefe e o cliente não vão esperar dois meses por cada nova alteração que apareça. Pior ainda se depois de dois meses de espera o sistema ainda não funcionar da maneira que deveria funcionar. Mas como criar um projeto flexível? O que você deve fazer para criar um grande software? Por onde começar? É sobre isto que vou falar.

Resumo DevMan

De que trata o artigo:

O artigo apresenta três simples etapas que irão te ajudar a criar um grande software orientado a objetos. Descubra por onde começar um novo projeto ou uma nova opção sem atrapalhar todo o resto do processo. Aprenda também alguns conceitos de UML e como ler um diagrama de classe.

Para que serve:

Começar um novo projeto ou uma alteração do jeito certo aumenta suas chances de atingir o objetivo proposto pelo software. Dificilmente você conseguirá criar grandes softwares se você não começar por onde deve.

Em que situação o tema é útil:

Este artigo ajudará você a aperfeiçoar seu processo de criação dos projetos. Ilustra as dificuldades em implementar tais tarefas, porém exibindo que esta apenas é uma questão de saber o que deve ser feito em cada fase do desenvolvimento.

Criando um grande software:

Por onde começar um programa? É muito comum começar um novo projeto ou até mesmo uma grande alteração de um projeto do ponto errado e depois perder muito tempo por conta disto. Você já tem experiência com programação, conhece os princípios da orientação a objetos e usa os padrões de projeto, mas mesmo assim você ainda perde muito tempo refazendo partes do seu projeto. Isto acontece, na maioria das vezes, porque você está começando um projeto, ou uma nova opção, de maneira errada. Qual a primeira coisa que você deve fazer em um programa? É sobre isto que você vai ler neste artigo. Através de um exemplo, você vai entender por onde começar e por que é tão importante começar do jeito certo.

Um grande Software

Todos reconhecem um grande Software e todos querem criar Softwares desta categoria. Acontece que o termo 'grande Software' pode ser interpretado de maneiras diferentes por cada pessoa. O que é realmente um grande Software? Cada pessoa pode ter sua ideia do que isto significa. Se você perguntar para o seu patrão, que não entende nada de programação, o que você precisa fazer para criar um grande Software ele provavelmente dirá:

• O que é um 'grande Software' para o seu chefe

Para criar um grande Software você deve criar um programa que atenda o objetivo do cliente exatamente da maneira como ele esperava que fosse e mesmo quando as coisas mudarem ou não acontecer como era esperado o sistema não deve falhar.

Por outro lado se fizer esta mesma pergunta para um programador apaixonado pela programação orientada a objetos ele poderá responder algo similar a isto:

• O que é um 'grande Software' para um programador

Você cria um grande Software quando cria um projeto orientado a objetos, fazendo com que cada objeto seja o único responsável por seu estado. Os objetos devem estar levemente interligados para tornar seu projeto aberto para expansão e fechado para modificações além de aplicar princípios e padrões de projetos já comprovados.

Para você qual destas duas respostas está correta? O que é um grande Software para você? As duas respostas estão certas, pois as duas formas de pensar juntas é o que formam um grande Software. Se você fizer o que seu chefe e o programador apaixonado por OO desejam, você terá criado um grande Software. Infelizmente não existe uma fórmula matemática para isto, mas existem algumas dicas que quando seguidas ajudam você a atingir o objetivo, criando um Software apreciado por todos. Três simples etapas é o que você precisa para aumentar suas chances de criar um grande projeto. Veja:

- Garanta que o sistema faz o que cliente espera que ele faça;
- Use princípios da programação orientada a objetos;
- Dedique-se a tornar o projeto reutilizável.

Então, neste momento alguém pode perguntar: "Se eu não seguir estas etapas não criarei um grande Software?"; "Por que exatamente estas etapas?". As duas perguntas são válidas e muito interessan-

e software

por todos

RAFAEL STAVARENGO

tes. É claro que estas etapas NÃO são uma regra. Na informática pessoas diferentes podem e geralmente encontram soluções diferentes para o mesmo problema. Por isto as três etapas que apresento anteriormente não são uma regra. Você pode encontrar outras maneiras de se criar um grande projeto.

Agora a segunda pergunta: "Por que exatamente estas etapas?". Se você quer criar um grande projeto, a primeira coisa a ser feita é conquistar seu cliente. Para isto você precisa acima de tudo atingir o objetivo do cliente. Por esta razão o seu patrão estava certo e esta também é a razão da primeira etapa. De nada serve um programa espetacular, cheio de recursos fantásticos, se "aquela pequena parte" que o cliente usa não funciona como ele pediu. Você deve criar programas que façam o que os clientes esperam até mesmo quando algo errado acontecer. Quando as coisas não funcionarem da maneira como esperado, o Software não pode deixar as coisas saírem do controle. Por outro lado o programador apaixonado por orientação a objetos também estava certo. O que justifica as últimas duas etapas. Depois que o Software estiver fazendo o que o cliente deseja que ele faça, é a hora de trabalhar no seu projeto para torná-lo reutilizável e flexível.

Você só irá conquistar seus clientes quando seu Software fizer o que eles desejam. Portanto a primeira etapa para se criar um grande projeto é garantir a satisfação do cliente. Depois de atingido este objetivo você pode dispensar um pouco mais de tempo para aplicar um pouco de princípios de orientação a objetos e por fim usar os padrões de projetos para tornar o mesmo mais flexível, facilitando as mudanças do dia-a-dia. Isto pode parecer complicado agora, mas no decorrer deste artigo você verá como alguns princípios simples podem mudar o ponto de vista dos clientes para com seus Softwares.

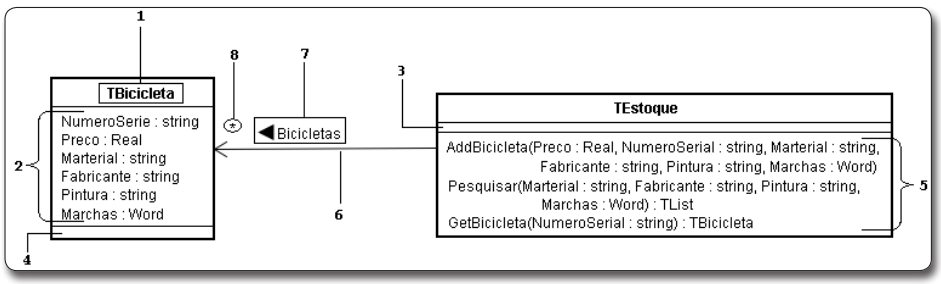


Figura 1. Design das classes

Imagine um cliente

A seguir você verá como aplicar as três etapas e como elas podem ajudar você a criar um grande projeto orientado a objetos. Para começar, é preciso que exista um cliente. O nome desta cliente imaginária é Mary. Ela possui uma loja de bicicletas e está a procura de um Software que ajude seus clientes a encontrar a bicicleta que procuram. Uma rápida conversa com Mary foi o suficiente para descobrir o que ela espera do novo Software.

• **Do que você precisa, Mary?**

Preciso de um programa que meus clientes possam usar para encontrar as bicicletas que procuram, sem minha ajuda. Os meus clientes normalmente vêm até a loja sabendo pouquíssimo sobre a bicicleta que desejam. Na maioria das vezes eles sabem apenas o fabricante. Claro que existem aqueles que não sabem nem o fabricante, então eles vêm até minha loja sabendo somente quantas marchas desejam ou a cor e o material do quadro. Neste caso fica mais difícil encontrar o que eles procuram, pois existem muitas bicicletas diferentes que são feitas do mesmo material, mas mesmo assim eu sempre encontro o que eles procuram.

Mary não precisa de nada complicado. O programa fará consultas em um estoque para encontrar as bicicletas que os clientes desejam. Logo, é possível deduzir que existiriam duas classes neste Software. Uma classe para representar as bicicletas e outra para representar o estoque onde estará a lista das bicicletas existentes. As bicicletas possuem poucas características: além das características que Mary citou fabricante, quantidade de marchas, cor e material do quadro, podemos deduzir também que cada bicicleta tem um número de série e um preço. Veja o diagrama de classes na **Figura 1**.

A classe TBicicleta terá todas as características que Mary citou, além do atributo NumeroSerie e Preco, ela não citou estes dois atributos, mas com certeza eles são necessários. O fabricante, a quantidade de marchas, o material e a pintura do quadro são as especificações de cada bicicleta. A classe TEstoque é a responsável por manter a lista de todas as bicicletas que Mary

vende. O método AddBicicleta é usado para adicionar uma nova bicicleta na lista. Já os métodos Pesquisar e GetBicicleta são usados para encontrar as bicicletas dentro da lista do estoque.

Entendendo a UML

A imagem que você vê na **Figura 1** é um diagrama de classes. Um diagrama de classes oferece uma visão geral das classes de um sistema, sem se preocupar com os detalhes de como elas são feitas. Isto significa que um diagrama de classes não mostra uma classe exatamente como ela está escrita no código fonte, até porque este diagrama não foi criado com este objetivo. O diagrama de classes facilita o diálogo com outros programadores e usuários. Com ele é possível dizer rapidamente qual o objetivo de uma classe sem a necessidade de navegar por dezenas de centenas de linhas de código, seja qual for a linguagem.

Eu enumerei algumas partes do diagrama na **Figura 1** para ajudar a entender o que significa todo o conteúdo dentro do diagrama de classes. O número 01 é o nome da classe. Este nome sempre estará em negrito. Em algumas situações ele pode estar em itálico para indicar uma classe abstrata. Cada classe representada no diagrama possui três partes distintas. A primeira parte você já conheceu: é o retângulo onde fica o nome da classe. Logo abaixo está a segunda parte – destacada pelos números 02 e 03, onde os atributos das classes são exibidos. No caso da classe TBicicleta existem seis atributos. A terceira parte de uma classe destacada pelos números 04 e 05, é onde os métodos são escritos. Como você pode ver a classe TBicicleta não possui métodos e a classe TEstoque possui 03 métodos.

O número 06 está destacando uma linha sólida que aponta de TEstoque para TBicicleta. Neste tipo de diagrama, uma linha sólida indica que uma classe está associada à outra seja por referência, herança ou qualquer outra tipo. Normalmente cada tipo de associação é feito com um tipo de linha diferente.

O símbolo destacado pelo número 8 indica a multiplicidade da associação (assim como nas tabelas do banco). O asterisco significa que a multiplicidade de 'Bicicletas' (texto destacado pelo número 07 é ilimitado). Isto significa que o estoque pode armazenar um número ilimitado de bicicletas. Além disto, o tipo de associação entre estas duas classes indica que a classe TEstoque tem um atributo chamado Bicicletas que vai armazenar uma lista de bicicletas. Repare que este atributo não aparece na classe TEstoque, mas a existência dele se torna implícita devido à associação com a classe TBicicleta.

Primeira versão do Software

O digrama de classe mostra um programa simples de codificar, portanto é hora de começar. No Delphi 2010 inicie um novo projeto. Salve-o com o nome de Exemplo.dpr. Você deve fechar sem salvar a unit do formulário que o Delphi cria automaticamente. Ela não é necessária agora. Para começar vou criar as classes TBicicleta e TEstoque. Crie uma nova unit através da opção File> New> Unit. Salve esta unit com o nome Bicicleta.pas. Em seguida digite o código que aparece na **Listagem 1**. A única diferença da **Listagem 1** para o diagrama da classe TBicicleta na **Figura 1** é o construtor declarado na linha 12 da **Listagem 1**. O construtor recebe todas as informações necessárias para criar uma bicicleta e alimenta os atributos com elas.

Crie outra unit, salve-a com o nome Estoque.pas. Nesta unit você deve digitar o código das **Listagens 2** e **3**. Na **Listagem 2** você vê a implementação do construtor e destrutor da classe. O construtor de TEstoque é usado para criar a variável Bicicletas que está declarado na linha 07. Está variável que é criada na linha 20, será usada para armazenar todas as bicicletas

Nota do DevMan

UML – Unified Modeling Language (Linguagem de modelagem unificada) – é uma especificação criada pela Rational Software que hoje é gerenciada pelo OMG – Object Management Group – mas na época em que a UML estava nascendo era liderada por Grady Booch, Jim Rumbaugh e Ivar Jacobson.

Nota do DevMan

Olhares mais atentos devem ter percebido que o construtor da classe TBicicleta – implementado na linha 12 da **Listagem 1** não invoca o construtor da classe TObject, de quem TBicicleta herda. Se isto fosse feito a linha de código **inherited Create**; estaria presente neste construtor. Ele não é invocado porque o construtor da classe TObject não faz nada, então invocá-lo é opcional e não faz diferença.

Listagem 1. Classe TBicicleta

```
01  unit Bicicleta;
02  interface
03  type
04    TBicicleta = class
05    public
06      NumeroSerie: string;
07      Preco: Real;
08      Material: string;
09      Fabricante: string;
10      Pintura: string;
11      Marchas: Byte;
12      constructor Create(Preco: Real; NumeroSerie, Material, Fabricante,
13        Pintura: string; Marchas: Byte); reintroduce;
14    end;
15  implementation

16  constructor TBicicleta.Create(Preco: Real; NumeroSerie, Material,
      Fabricante, Pintura: string; Marchas: Byte);
17  begin
18    Self.NumeroSerie := NumeroSerie;
19    Self.Preco := Preco;
20    Self.Material := Material;
21    Self.Fabricante := Fabricante;
22    Self.Pintura := Pintura;
23    Self.Marchas := Marchas;
24  end;

25  end.
```

do estoque, ou seja, a pesquisa será feita com base nas bicicletas existentes na lista da variável "Bicicletas".

Nas linhas 09, 11 e 12 da **Listagem 2** está a declaração dos métodos AddBicicleta, GetBicicleta e Pesquisar respectivamente. Estes são os métodos de manipulação do estoque. É através destes métodos que outras classes vão interagir com a lista de bicicletas cadastradas. A implementação destes três métodos encontra-se na **Listagem 3**.

O método AddBicicleta, cuja implementação começa na linha 01 da **Listagem 3** é usado para cadastrar uma bicicleta no estoque. Este método recebe todas as informações necessárias para criar uma bicicleta, o que acontece na linha 06. Após criar uma bicicleta ela é adicionada ao estoque: isto acontece na linha 08.

O método GetBicicleta, cuja implementação aparece na linha 10 da **Listagem 3** é usado para buscar por uma bicicleta específica. Ele recebe o número de série da bicicleta e procura por ela no estoque. Se não encontrar ele retorna **nil**. Lembre que o número de série é único para cada bicicleta.

Embora os métodos Pesquisar e GetBicicleta façam a mesma coisa, ou seja, procurar por bicicletas, eles possuem objetivos diferentes. Enquanto que GetBicicleta procura por uma única bicicleta, o método Pesquisar que pode ser visto na linha 24 da **Listagem 3**, procura por todas as bicicletas do estoque que possuem características específicas. Este é o método que será invocado quando os clientes de Mary procurarem pelas bicicletas que desejam. A ideia do método Pesquisar é simples: você entra com as características que deseja em uma bicicleta e ele retorna para você todas as bicicletas do estoque que se enquadram com o que você pediu. Veja na linha 30 que é feita uma iteração com todas as bicicletas

Listagem 2. Primeira parte da classe TEstoque

```
01  unit Estoque;

02  interface

03  uses Generics.Collections, Bicicleta, SysUtils;

04  type
05   TEstoque = class
06   private
07    Bicicletas: TList<TBicicleta>;
08   public
09    procedure AddBicicleta(Preco: Real; NumeroSerie, Material, Fabricante,
10     Pintura: string; Marchas: Byte);
11    function GetBicicleta(NumeroSerie: string): TBicicleta;
12     function Pesquisar(Material, Fabricante, Pintura: string; Marchas: Byte)
13      : TList<TBicicleta>;
14    constructor Create; reintroduce;
15    destructor Destroy; override;
16   end;

17  implementation

18  constructor TEstoque.Create;
19  begin
20   Bicicletas := TList<TBicicleta>.Create;
21  end;

22  destructor TEstoque.Destroy;
23  begin
24   FreeAndNil(Bicicletas);
25   inherited;
26  end;

27  { Implementação dos outros métodos }

28  end.
```

Listagem 3. Segunda parte da classe TEstoque

```
01  procedure TEstoque.AddBicicleta(Preco: Real; NumeroSerie, Material, Fabricante,
02   Pintura: string; Marchas: Byte);
03  var
04   Bicicleta: TBicicleta;
05  begin
06   Bicicleta := TBicicleta.Create(Preco, NumeroSerie, Material, Fabricante,
07    Pintura, Marchas);
08   Bicicletas.Add(Bicicleta);
09  end;

10  function TEstoque.GetBicicleta(NumeroSerie: string): TBicicleta;
11  var
12   Bicicleta: TBicicleta;

13  begin
14   Result := nil;
15   for Bicicleta in Bicicletas do
16   begin
17    if NumeroSerie = Bicicleta.NumeroSerie then
18    begin
19     Result := Bicicleta;
20     Exit;
21    end;
22   end;
23  end;

24  function TEstoque.Pesquisar(Material, Fabricante, Pintura: string;
25   Marchas: Byte): TList<TBicicleta>;
26  var
27   Bicicleta: TBicicleta;
28  begin
29   Result := TList<TBicicleta>.Create;
30   for Bicicleta in Bicicletas do
31   begin
32    if (Material <> '') and (Material <> Bicicleta.Material) then
33     Continue;
34    if (Fabricante <> '') and (Fabricante <> Bicicleta.Fabricante) then
35     Continue;
36    if (Pintura <> '') and (Pintura <> Bicicleta.Pintura) then
37     Continue;
38    if (Marchas > 0) and (Marchas <> Bicicleta.Marchas) then
39     Continue;

40    Result.Add(Bicicleta);
41   end;
42  end;
```

cadastradas, e para cada uma é verificado se as características se enquadram com as procuradas (você pode ver estes testes entre as linhas 32 e 39). A instrução **continue** que pode ser vista nas linhas 33, 35, 37 e 39 faz com que o loop passe para o próximo item. Por fim, na linha 40, caso as características sejam compatíveis com o que foi procurado, a bicicleta é adicionada à variável de retorno.

Alimentando o estoque

As classes necessárias para o programa funcionar já foram criadas. Agora está na hora de criar o formulário que será usado para fazer as pesquisas. Crie um novo formulário através da opção File> New>Form e salve a unit deste formulário com o nome Principal.pas. Neste formulário estará a interface para que os clientes de Mary pesquisem pelas bicicletas que desejam comprar. Depois de salvar, adicione e configure os componentes de acordo com a **Tabela 1** e em seguida deixe o layout do formulário semelhante ao apresentado na **Figura 2**.

Para testar o programa será necessário que existam algumas bicicletas cadastradas. Este cadastro vai acontecer quando o formulário for criado. Como você pode ver na **Listagem 4**, no evento FormCreate (linha 09), um objeto da classe TEstoque é criado, e logo em seguida nas linhas 12, 13, 14 e 15 quatro novas bicicletas são adicionadas. Isto significa que após abrir o programa o estoque terá quatro bicicletas cadastradas. Este cadastro é feito invocando o método AddBicicleta do estoque. A variável Estoque está declarada na linha 05.

O algoritmo de pesquisa já está pronto, como você viu na **Listagem 3**. Será fácil fazer o formulário de pesquisa mostrar as bicicletas retornadas pelo método Pesquisar. A pesquisa será feita quando o usuário clicar no botão "BotaoPesquisar", portanto o código da **Listagem 5** mostra o que acontece no evento OnClick deste botão. Primeiro, na linha 07, o método Pesquisar do Estoque é invocado (o objeto Estoque foi criado na **Listagem 4**, lembra?). O resultado da pesquisa é armazenado na variável BikesEncontradas (adotamos a palavra Bike para o nome não ficar muito grande). Com o resultado da pesquisa em mãos, basta fazer um loop para cada bicicleta encontrada, adicionando as informações sobre elas no Memo1 do formulário. Este loop começa na linha 16, mas

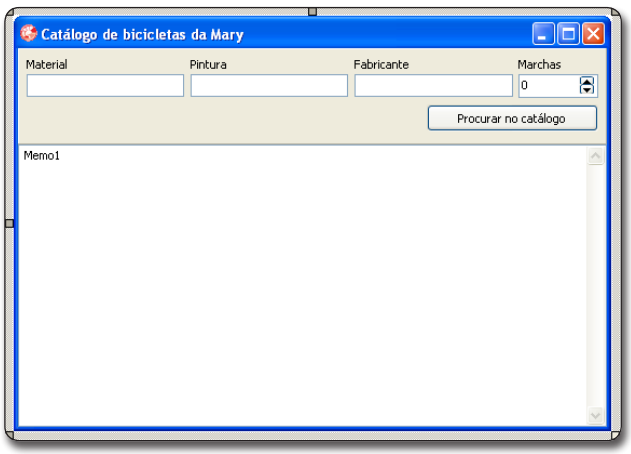


Figura 2. Layout do formulário de pesquisa

Nota do DevMan

Você percebeu que a variável "Bicicletas" declarada na linha 07 da **Listagem 2** é privada e a única maneira de manipulá-la é através dos métodos AddBicicleta, GetBicicleta e Pesquisar? Isto é o encapsulamento em ação. Encapsulamento é um dos fundamentos básicos da programação orientada a objetos. Encapsular é o ato de proteger algo, para que outras classes não o acessem diretamente. Você pode encapsular muito mais do que somente atributos, como verá até o final do artigo.

Componente	Propriedade	Valor
TForm	Name	Form1
	Caption	Catálogo de bicicletas da Mary
TLabel	Caption	Material
TEdit	Name	EdMaterial
TLabel	Caption	Pintura
TEdit	Name	EdPintura
TLabel	Caption	Fabricante
TEdit	Name	EdFabricante
TLabel	Caption	Marchas
TSpinEdit	Name	EdMarchas
TButton	Name	BotaoPesquisar
	Caption	Procurar no catálogo
TMemo	Name	Memo1
	ReadOnly	True
	ScroolBars	ssVertical

Tabela 1. Componentes do formulário de pesquisa

Listagem 4. Alimentando o estoque

```
01  type
02    TForm1 = class(TForm)
03      { Componentes do formulário }
04    private
05      Estoque: TEstoque;
06    public
07    end;

08  implementation

09  procedure TForm1.FormCreate(Sender: TObject);
10  begin
11    Estoque := TEstoque.Create;
12    Estoque.AddBicicleta(125.5, '8536303956', 'Alumínio', 'Dorat', 'Cromada', 21);
13    Estoque.AddBicicleta(191.85, '9788576083382', 'Ferro', 'Dorat', 'Simples', 8);
14    Estoque.AddBicicleta(1098.9, '9788520431047', 'Fibra de carbono', 'Fitting',
        'Camaleão', 21);
15    Estoque.AddBicicleta(918.9, '9788535238693', 'Alumínio', 'Acto', 'Cromada', 21);
16  end;

17  procedure TForm1.FormDestroy(Sender: TObject);
18  begin
19    FreeAndNil(Estoque);
20  end;
```

ele só será executado se o método Pesquisar retornar alguma bicicleta, pois se nenhuma bicicleta for retornada a instrução da linha 12 é executada.

Um teste rápido

O programa está pronto, mas antes de enviá-lo para Mary, é preciso testar para garantir que tudo está funcionando. Então para testar, eu executei o programa, e pesquisei por algumas bicicletas para garantir que o resultado estava de acordo. Veja na **Figura 3** o resultado de um dos testes que fiz.

No mundo real...

Mary recebeu o programa e ficou contente, porém pouco tempo depois começou a perder vendas. A reclamação dela é que os clientes não encontram o que procuraram mesmo quando a bicicleta existe no estoque. Veja na **Figura 4** um exemplo passado pela Mary.

Como isto pode acontecer? O programa foi testado e tudo estava funcionando bem. Acontece que no 'mundo real' as coisas quase nunca são feitas como se espera. Neste caso, mesmo existindo duas bicicletas de alumínio, o Catálogo da Mary não está encontrando nada e os clientes estão comprando em outras lojas.

O problema é simples: no estoque existem bicicletas cujo quadro é feito de "Alumínio" e não de "alumínio". Alguns podem pensar: "Ufa! Que susto, eu imaginei que o programa tinha um erro!". Eu diria para esta pessoa: "Seu programa tem um erro. Ele não faz o que o cliente espera.". Mary espera que seus clientes encontrem pelas bicicletas do estoque, e o Software não está fazendo isto, pelo contrário, por culpa do Software Mary está perdendo clientes. De fato, você poderia dizer que a culpa não é sua, mas sim dos clientes que digitaram errado, só que isto não muda o fato de que os clientes continuam comprando em outra loja.

A primeira etapa para criar um grande Software diz: "O Software deve fazer o que o cliente espera que ele faça", e o Catálogo da Mary não faz o que ela espera. Isto aconteceu porque eu comecei o projeto de forma errada. Você deve começar um novo projeto se preocupando em garantir que o projeto fará exatamente o que o usuário quer. Como eu "queimei" esta etapa, terei de voltar à estaca zero: mexer no código fonte para corrigir o erro. O problema está no método Pesquisar da classe TEstoque (linha 24, **Listagem 3**). Veja neste método que a pesquisa é feita sem levar em consideração se o termo pesquisado está em caixa alta ou não.

Solução: as três etapas

Veja o que foi feito até agora. Este projeto está horrível com este monte de strings espalhadas por todos os lados. Talvez fosse melhor usar constantes ou tipos enumerados para o material, pintura e fabricante das bicicletas. Além disto, o projeto está muito longe do que pode ser considerado reutilizável, pois as classes TEstoque e TBicicleta dependem muito uma da outra.

Espere um pouco! A primeira etapa diz que devemos nos concentrar no que o cliente deseja, e as soluções do último parágrafo só

Nota

Faltam várias linhas de códigos na **Listagem 4**, como por exemplo, a declaração de cada componente do formulário, que deveria estar entre as linhas 02 e 04 (não são relevantes aqui). Além disto, também está faltando a sessão uses, mesmo assim você não deve se esquecer de adicionar a unit Estoque.pas na sessão uses da unit Principal.pas.

Nota

Para a **Listagem 5** funcionar, você não pode se esquecer de declarar as units Bicicleta.pas e Generics.Collections.pas na sessão uses.

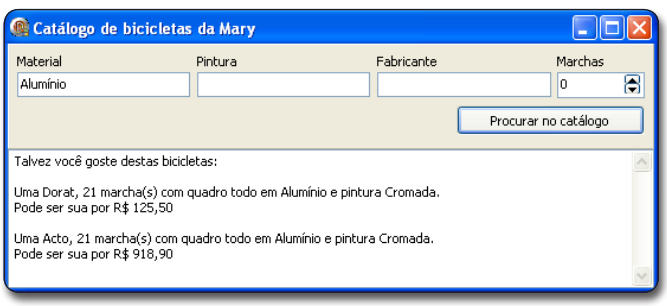


Figura 3. Procurando por bicicletas de alumínio

Figura 4. Mas existem bicicletas de alumínio!

Listagem 5. Mostrando o resultado da pesquisa

```
01  uses Bicicleta, Generics.Collections;

02  procedure TForm1.BotaoPesquisarClick(Sender: TObject);
03  var
04    BikesEncontradas: TList<TBicicleta>;
05    Bike: TBicicleta;
06  begin
07    BikesEncontradas := Estoque.Pesquisar(EdMaterial.Text, EdFabricante.Text,
08      EdPintura.Text, EdMarchas.Value);

09    Memo1.Clear;
10    if BikesEncontradas.Count = 0 then
11    begin
12      Memo1.Lines.Add('Desculpe, não temos nada para você hoje!');
13    end
14    else
15    begin
16      Memo1.Lines.Add('Talvez você goste destas bicicletas:');
17      for Bike in BikesEncontradas do
18      begin
19        Memo1.Lines.Add('');
20        Memo1.Lines.Add('Uma ' + Bike.Fabricante + ',' + IntToStr(Bike.Marchas) + '
          marcha(s) com quadro todo em ' + Bike.Material + ' e pintura ' + Bike.Pintura + '.');
21        Memo1.Lines.Add('Pode ser sua por ' + FormatCurr('R$ ##.00', Bike.Preco));
22      end;
23    end;
24  end;
```

dizem respeito ao que o projeto precisa para melhorar e não no que o cliente quer. A única coisa necessária para atender os requisitos do cliente é converter todas as strings para minúscula ou maiúscula antes de compará-las, mas se o projeto já está terrível imagine como será se existirem conversões de strings espalhadas por todos os lados. Não custa nada mexer um pouco no projeto para evitar problemas que com certeza resultarão em dores de cabeça mais tarde.

Isto significa que mesmo durante a primeira das três etapas é possível se preocupar um pouco com o projeto, pois você quer que as partes do seu Software estejam no lugar certo antes de precisar fazer grandes mudanças. Aplicar princípios da orientação a objetos ainda durante a primeira etapa é comum, e necessário, desde que você não perca o foco de atender os requisitos do cliente.

Para começar a corrigir o problema do Catálogo da Mary, eu vou aproveitar e mudar um "pouquinho" o projeto, para tentar me livrar de todas as strings espalhadas pelo código. Para conseguir isto usarei os tipos enumerados, que permitem definir quais são os únicos valores aceitos para uma variável. Desta maneira o programa fica protegido contra os dados inválidos. O único problema dos tipos enumerados do Delphi é que eles não permitem definir um nome para cada tipo. Na apresentação disponível na versão digital desta edição, eu apresento uma maneira de simular o que é chamado de "enum" nas demais linguagens, para suprir as necessidades dos tipos enumerados existentes no Delphi.

A **Listagem 6** mostra um pedaço da unit Bicicleta.pas. Eu escolhi declarar os tipos enumerados nesta unit, pois apesar deles serem usados em mais partes do projeto, na verdade eles dizem respeito apenas às bicicletas: eles são tipos que a bicicleta precisa para existir. Veja a declaração destes tipos entre as linhas 02 e 04. Apenas fazer a declaração não é o suficiente, é necessário usá-los. É a classe TBicicleta quem precisa deles, por isto os atributos Material, Fabricante e Pintura da classe TBicicleta mudaram do tipo string para o respectivo tipo enumerado (veja nas linhas 09, 10 e 11). Depois de mudar o tipo dos atributos das bicicletas vários erros de compilação acontecem no projeto. O primeiro deles no construtor da própria classe TBicicleta, pois os parâmetros do construtor ainda são do tipo string. Basta alterar o tipo dos parâmetros, como mostrado na linha 13 da **Listagem 6** e a classe TBicicleta ficará livre de erros de compilação. Mas ainda existem outras partes do projeto que ainda precisam ser alteradas para usar os novos tipos enumerados.

Depois de criar os tipos enumerados, já é possível corrigir o método pesquisar da classe TEstoque. Na verdade, eu vou alterar a assinatura deste método. A intenção é deixá-lo mais simples, pois receber este monte de parâmetros não parece uma boa prática. Vou alterar o método Pesquisar para que ele receba um objeto da classe TBicicleta, ao invés de uma lista de parâmetros referentes às características das bicicletas.

Veja na linha 12 da **Listagem 7** a nova implementação do método Pesquisar, que agora está corrigida e não fará que a Mary perca mais clientes. A diferença da antiga para a nova implementação é basicamente a forma de busca, pois agora a pesquisa não depende da maneira como o usuário digita os termos procurados, mas sim dos tipos enumerados criados na **Listagem 6**. Além das alterações feitas nos parâmetros

Listagem 6. Criando os tipos enumerados

```
01  type
02    TFabricante = (fNaoDefinido, fDorat, fFitting, fActo);
03    TMaterial = (mNaoDefinido, mAluminio, mFerro, mCarbono);
04    TPintura = (pNaoDefinido, pCromada, pCamaleao, pSimples);
05    TBicicleta = class
06    public
07      NumeroSerie: string;
08      Preco: Real;
09      Material: TMaterial;
10      Fabricante: TFabricante;
11      Pintura: TPintura;
12      Marchas: Byte;
13      constructor Create(Preco: Real; NumeroSerie: string; Material: TMaterial;
        Fabricante: TFabricante; Pintura: TPintura; Marchas: Byte); reintroduce;
14    end;
```

Listagem 7. Corrigindo a classe TEstoque

```
01  type
02    TEstoque = class
03    private
04      Bicicletas: TList<TBicicleta>;
05    public
06      procedure AddBicicleta(Preco: Real; NumeroSerie: string; Material: TMaterial;
        Fabricante: TFabricante; Pintura: TPintura; Marchas: Byte);
07      function GetBicicleta(NumeroSerie: string): TBicicleta;
08      function Pesquisar(Bike: TBicicleta): TList<TBicicleta>;
09      constructor Create; reintroduce;
10      destructor Destroy; override;
11    end;

12  function TEstoque.Pesquisar(Bike: TBicicleta): TList<TBicicleta>;
13  var
14    Bicicleta: TBicicleta;
15  begin
16    Result := TList<TBicicleta>.Create;
17    for Bicicleta in Bicicletas do
18    begin
19      if (Bike.Material <> mNaoDefinido) and (Bike.Material <> Bicicleta.Material) then
20        Continue;
21      if (Bike.Fabricante <> fNaoDefinido) and (Bike.Fabricante <>
        Bicicleta.Fabricante) then
22        Continue;
23      if (Bike.Pintura <> pNaoDefinido) and (Bike.Pintura <> Bicicleta.Pintura) then
24        Continue;
25      if (Bike.Marchas > 0) and (Bike.Marchas <> Bicicleta.Marchas) then
26        Continue;

27      Result.Add(Bicicleta);
28    end;
29  end;
```

e na implementação do método Pesquisar, a assinatura do método AddBicicleta também foi alterada (somente a assinatura). Confira na **Listagem 7**.

As alterações das **Listagens 6** e **7** são o suficiente para atingir os requisitos do cliente. Mas com estas alterações a unit Principal.pas parou de compilar, por isto ela também precisa ser alterada. Agora os, fabricantes, as pinturas e o material de cada bicicleta já estão definidos e não faz mais sentido usar TEdit para digitar estas informações. O TRadioGroup é o ideal para esta tarefa. Apague todos os TEdits, adicione três novos TRadioGroups e configure-os de acordo com as propriedades exibidas na **Tabela 2**. Na **Figura 5** está o novo layout do formulário.

Trocar os componentes do formulário não fará que a unit volte a compilar, para isto o código da classe TForm1 deve ser corrigido. Somente o evento o método BotaoPesquisarClick precisa ser alterado. É uma alteração pequena, pois antes o método Pesquisar da classe TEstoque recebia strings como parâmetros, mas agora ele recebe um objeto da classe TBicicleta. A correção deste problema aparece nas linhas 06 e 07 da **Listagem 8**. Os métodos GetMaterial, GetFabricante e GetPintura usados na linha 06, logo serão apresentados a você. Outra alteração está na linha 19 onde foi necessário converter os tipos enumerados para strings antes de apresentá-los ao usuário. Nesta linha aparecem mais três métodos novos, são eles: GetMaterialStr, GetFabricanteStr e GetPinturaStr.

Como agora os atributos Fabricante, Material e Pintura são tipos enumerados, é necessário convertê-los, por isto eu precisei criar seis novos métodos para me ajudar nesta tarefa. Três deles convertem o tipo enumerado em uma string e os outros três convertem o valor selecionado no TRadioGroup para o tipo enumerado correto. Veja nas **Listagens 9** e **10** estes novos métodos, que dispensam explicações. Na **Listagem 9**, estão os métodos usados para a opção que o usuário selecionou no TRadioGroup para o tipo enumerado correspondente, e na **Listagem 10** está a conversão dos tipos enumerados para uma string.

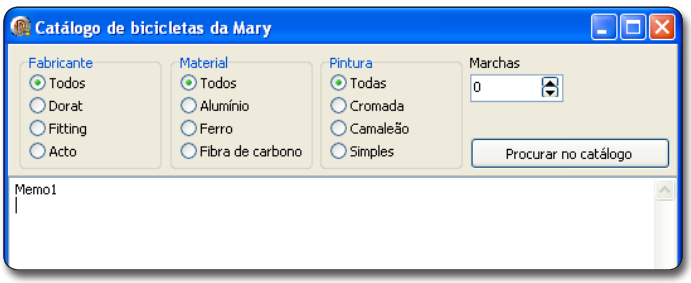


Figura 5. Novo layout do formulário de pesquisa

Componente	Propriedade	Valor
TRadioGroup	Name	rgFabricante
	Caption	Fabricante
	ItemIndex	0
	Items	Todos Dorat Fitting Acto
TRadioGroup	Name	rgMaterial
	Caption	Material
	ItemIndex	0
	Items	Todos Alumínio Ferro Fibra de carbono
TRadioGroup	Name	rgPintura
	Caption	Pintura
	ItemIndex	0
	Items	Todas Cromada Camaleão Simples

Tabela 2. Novos componentes do formulário de pesquisa

Nota

Na versão digital desta edição está disponível uma apresentação onde eu mostro uma forma mais orientada a objetos de criar tipos enumerados. Assim você não vai precisar espalhar métodos extras pelo projeto apenas para converter os tipos enumerados em strings.

Listagem 8. Corrigindo a classe TForm1

```
01  procedure TForm1.BotaoPesquisarClick(Sender: TObject);
02  var
03    BikesEncontradas: TList<TBicicleta>;
04    Bike: TBicicleta;
05  begin
06    Bike := TBicicleta.Create(0, '', GetMaterial, GetFabricante, GetPintura, EdMarchas.
      Value);
07    BikesEncontradas := Estoque.Pesquisar(Bike);
08    Memo1.Clear;
09    if BikesEncontradas.Count = 0 then
10    begin
11      Memo1.Lines.Add('Desculpe, não temos nada para você hoje!');
12    end
13    else
14    begin
15      Memo1.Lines.Add('Talvez você goste destas bicicletas:');
16      for Bike in BikesEncontradas do
17      begin
18        Memo1.Lines.Add('');
19        Memo1.Lines.Add('Uma ' + GetFabricanteStr(Bike.Fabricante) + ', ' + IntToStr
          (Bike.Marchas) + ' marcha(s) com quadro todo em ' + GetMaterialStr
          (Bike.Material) + ' e pintura ' + GetPinturaStr(Bike.Pintura) + '.');
20        Memo1.Lines.Add('Pode ser sua por ' + FormatCurr('R$ ##.00', Bike.Preco));
21      end;
22    end;
23  end;
```

Por último, precisamos apenas mudar o método OnCreate do formulário para trabalhar agora com tipos enumerados no lugar de strings, isto fará com que ele fique da seguinte maneira:

```
Estoque := TEstoque.Create;
Estoque.AddBicicleta(125.5, '8536303956',
  mAluminio, fDorat, pCromada, 21);
Estoque.AddBicicleta(191.85, '9788576083382',
  mFerro, fDorat, pSimples, 8);
Estoque.AddBicicleta(1098.9, '9788520431047',
  mCarbono, fFitting, pCamaleao, 21);
Estoque.AddBicicleta(918.9, '9788535238693',
  mAluminio, fActo, pCromada, 21);
```

Depois destas alterações o Catálogo da Mary está pronto e funciona como ela espera.

Não perca o foco

Não perca o foco, as alterações feitas até agora garantem apenas o funcionamento básico do Software, pois para criar um grande Software, a primeira coisa a se fazer é garantir que este faça aquilo que o cliente quer da maneira como ele quer. Você só passa para a segunda etapa quando o cliente estiver totalmente satisfeito.

Concluir a primeira etapa antes de começar a segunda é importe, pois até que o sistema esteja do jeito que o cliente espera você terá que fazer muitas alterações e tentar alterar muito o projeto antes de concluir a funcionalidade básica da aplicação resultará em um grande desperdício de tempo. Lembre-se que estas etapas oferecem um caminho mais simples de criar um grande projeto e nada obriga você a segui-las.

Mary recebeu a nova versão do catálogo e em pouco tempo suas vendas voltaram ao fluxo normal. Isto é muito bom, porque agora que o Software faz exatamente aquilo que o cliente espera. É hora de passar à segunda etapa: "use os princípios da programação orientada a objetos". É nesta etapa que você verifica se o programa que está funcionando foi criado de uma maneira que faça sentido. Agora você adiciona mais flexibilidade procurando por problemas de projeto e aplica os princípios da orientação a objetos para resolvê-los.

Segunda etapa

Como encontrar problemas em um projeto? Criar uma análise textual do problema vai ajudar você a descobrir se o projeto precisa ser alterado para fazer mais sentido, ou oferecer mais flexibilidade (como preferir dizer). Criar uma análise textual é a parte mais fácil: basta escrever um texto corrido explicando o que o Software, ou a parte que você está analisando, faz exatamente. Na **Listagem 11** exibimos uma análise textual do método Pesquisar.

O grande problema deste projeto é a má utilização de objetos. Cada objeto de um projeto OO deve ter apenas uma responsabilidade, que será executada da melhor maneira possível. Quando você usa objetos para executar mais de uma função, você está usando mal os seus objetos.

No Catálogo da Mary os clientes não informam o preço ou o número de série

Listagem 9. Convertendo a seleção do TRadioGroup

```
01  function TForm1.GetMaterial: TMaterial;
02  begin
03    case rgMaterial.ItemIndex of
04      1: Result := maluminio;
05      2: Result := mferro;
06      3: Result := mcarbono;
07    else Result := mNaoDefinido;
08    end;
09  end;

10  function TForm1.GetFabricante: TFabricante;
11  begin
12    case rgFabricante.ItemIndex of
13      1: Result := fdorat;
14      2: Result := fFitting;
15      3: Result := fActo;
16    else Result := fNaoDefinido;
17    end;
18  end;

19  function TForm1.GetPintura: TPintura;
20  begin
21    case rgPintura.ItemIndex of
22      1: Result := pCromada;
23      2: Result := pCamaleao;
24      3: Result := pSimples;
25    else Result := pNaoDefinido;
26    end;
27  end;
```

Listagem 10. Convertendo os tipos enumerados para string

```
01  function TForm1.GetPinturaStr(Pintura: TPintura): string;
02  begin
03    case Pintura of
04      pCromada: Result := 'Cromada';
05      pCamaleao: Result := 'Camaleao';
06      pSimples: Result := 'Simples';
07    else Result := '<pintura desconhecida>';
08    end;
09  end;

10  function TForm1.GetMaterialStr(Material: TMaterial): string;
11  begin
12    case Material of
13      maluminio: Result := 'Alumínio';
14      mcarbono: Result := 'Carbono';
15      mferro: Result := 'Ferro';
16    else Result := '<material desconhecido>';
17    end;
18  end;

19  function TForm1.GetFabricanteStr(Fabricante: TFabricante): string;
20  begin
21    case Fabricante of
22      fActo: Result := 'Acto';
23      fdorat: Result := 'Dorat';
24      fFitting: Result := 'Fitting';
25    else Result := '<fabricante desconhecido>';
26    end;
27  end;
```

Dica

Não se esqueça: criar uma análise textual do que o sistema faz, vai te ajudar descobrir se ele está fazendo o que precisa DA MANEIRA QUE DEVERIA. Em outras palavras, te ajuda a descobrir se a maneira como o sistema foi projetado faz sentido.

da bicicleta e mesmo assim o método Pesquisar espera por um objeto com estes atributos. O maior indício de que um objeto tem mais de uma função é quando você o usa sem precisar de todos os seus atributos.

Graças à análise textual (**Listagem 11**), descobrimos que os clientes informam apenas as especificações de bicicletas, mas o método Pesquisar está recebendo um objeto bicicleta, o qual possui mais atributos do que o que o método Pesquisar precisa.

Atualmente o método Pesquisar da classe TEstoque recebe uma bicicleta como parâmetro, porém nem todos os atributos da classe TBicicleta são usados para pesquisa. Em outras palavras, o método Pesquisar não procura por uma bicicleta, mas sim pelas especificações de uma bicicleta (como está escrito no item 1 da análise textual). O resultado disto é o mau uso dos objetos, pois estou usando um objeto bicicleta como se fosse as especificações de uma bicicleta. Veja a **Figura 6** para entender melhor.

A solução: um novo objeto

Já foi identificado, com a ajuda da análise textual, que o Software está usando bicicletas como se fossem especificações de bicicletas. Isto está acumulando responsabilidades sobre os objetos de TBicicleta, pois hora eles são usados como bicicletas de verdade e hora eles são usados apenas como especificações de bicicletas. A solução para isto é separar (encapsular), as especificações em uma nova classe. A primeira impressão é que isto vai gerar códigos duplicados, pois se no futuro, Mary começar a vender bicicletas com diferentes tipos de freios, será necessário criar um novo atributo tanto na classe bicicleta quanto na nova classe que possui apenas as especificações.

Isto SERIA um problema se não existisse o encapsulamento. Para evitar o código duplicado vamos encapsular os atributos que dizem respeito às especificações da bicicleta em outra classe. Veja na **Figura 7** como vai ficar o diagrama de classes do projeto depois de encapsular as especificações.

O encapsulamento não serve apenas para proteger os atributos de uma classe usando os famosos métodos Get e Set. O encapsulamento também pode ser usado para separar as partes do seu projeto, como foi usado no Catálogo da Mary. Criamos uma nova classe para suprir as necessidades de trabalhar apenas com as especificações das bicicletas. Desta maneira conseguimos evitar a má utilização dos objetos, pois não

Listagem 11. Análise textual do método Pesquisar

1. Cliente informa o que procura em uma bicicleta.
Cada cliente vai até a loja da Mary com as especificações que uma bicicleta deve ter: o material ou a cor do quadro, quantidade de marchas ou o fabricante. Eles passam estas informações para o software e ele busca pelas bicicletas do estoque.

2. O software faz a busca
O software faz um loop pelas bicicletas do estoque, e para cada bicicleta verifica se ela tem as características que o cliente deseja. Se encontrar alguma bicicleta, ela será adicionada na lista de bicicletas disponíveis para o cliente.

3. O cliente vê a lista das opções
Por fim, o cliente recebe como resposta uma lista com todas as bicicletas que possui as características que ele procura.

```
procedure TForm1.BotaoPesquisarClick(Sender: TObj
var
  BikesEncontradas: TList<TBicicleta>;
  Bike: TBicicleta;
begin
  Bike := TBicicleta.Create(0, '', GetMaterial, GetFabricante, GetPintura,
    EdMarchas.Value);
  BikesEncontradas := Estoque.Pesquisar(Bike);

  Memo1.Clear;
```

Criamos uma bicicleta, mas não usamos todos os seus atributos, pois na verdade não precisamos de um objeto bicicleta completo, mas sim das especificações de uma bicicleta.

Figura 6. Mau uso de um objeto

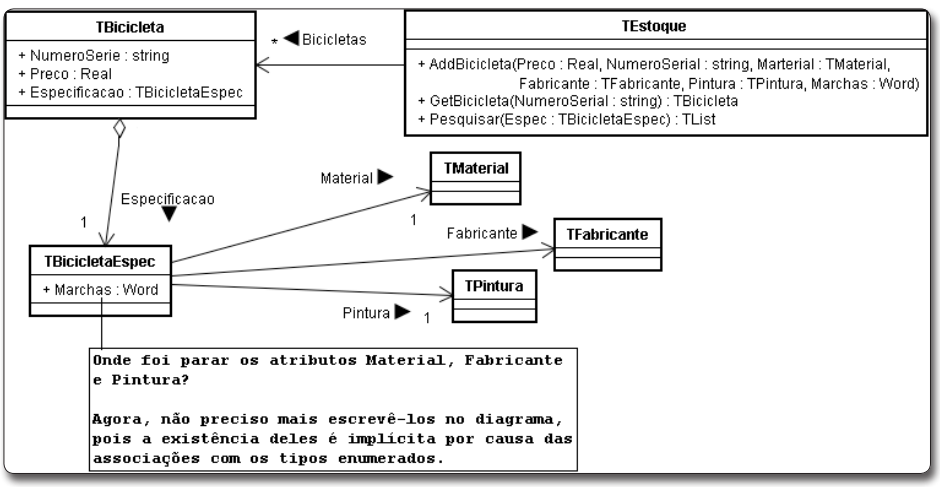


Figura 7. Encapsulando as especificações

Nota

Estamos visualizando apenas as partes mais relevantes do projeto, que sofrerão alteração por conta da nova classe TBicicletaEspec, mas você pode baixar o arquivo deste artigo onde estará todo o código do Catálogo da Mary.

Listagem 12. Metodo Pesquisar com a classe TBicicletaEspec

```
01  function TEstoque.Pesquisar(Espec: TBicicletaEspec): TList<TBicicleta>;
02  var
03    Bicicleta: TBicicleta;
04  begin
05    Result := TList<TBicicleta>.Create;
06    for Bicicleta in Bicicletas do
07    begin
08      if (Espec.Material <> mNaoDefinido) and (Espec.Material <>
        Bicicleta.Especificacao.Material) then
09        Continue;
10      if (Espec.Fabricante <> fNaoDefinido) and (Espec.Fabricante <>
        Bicicleta.Especificacao.Fabricante) then
11        Continue;
12      if (Espec.Pintura <> pNaoDefinido) and (Espec.Pintura <>
        Bicicleta.Especificacao.Pintura) then
13        Continue;
14      if (Espec.Marchas > 0) and (Espec.Marchas <>
        Bicicleta.Especificacao.Marchas) then
15        Continue;
16      Result.Add(Bicicleta);
17    end;
18  end;
```

Listagem 13. Invocando o novo metodo Pesquisar com a classe TBicicletaEspec

```
01  procedure TForm1.BotaoPesquisarClick(Sender: TObject);
02  var
03    BikesEncontradas: TList<TBicicleta>;
04    Bike: TBicicleta;
05    BikeSpec: TBicicletaEspec;
06  begin
07    BikeSpec := TBicicletaEspec.Create(GetMaterial, GetFabricante, GetPintura,
      EdMarchas.Value);
08    BikesEncontradas := Estoque.Pesquisar(BikeSpec);
09    { Restante do código foi cortado para economizar espaço}
```

precisamos criar uma bicicleta inteira para trabalhar apenas com suas especificações.

A **Listagem 12** mostra o novo método Pesquisar que agora está recebendo somente as especificações de uma bicicleta, ao invés de uma bicicleta completa. E na **Listagem 13** está o evento OnClick do botão de pesquisa. Como você pode ver na linha 07, o formulário está criando um objeto de TBicicletaEspec para fazer a busca. Isto faz mais sentido do que usar uma bicicleta inteira numa busca, sendo que algumas informações como preço e número de série, não são usadas pela pesquisa.

Terceira etapa

Depois de garantir que o Software faz o que cliente deseja e de aplicar bons princípios de orientação a objetos, você terá uma aplicação mais maleável, permitindo fácil manutenabilidade. Mas ainda falta a terceira etapa: nesta fase, você aplica os padrões de projetos verdadeiramente comprovados e ao final dela, você terá criado um grande Software que realmente atende as necessidades do mundo real como se espera.

Encontrar informações sobre padrões de projetos é uma tarefa fácil. A ClubeDelphi possui muitas edições com artigos sobre o assunto. É nesta fase, que você aplica estes conhecimentos. No caso do Catálogo da Mary eu poderia usar a delegação, passando a responsabilidade de comparação entre especificações para a classe TBicicletaEspec. Assim reduziria a quantidade de código do método Pesquisar e evitaria redundância de código no futuro.

Conclusão

O objetivo das três etapas é ajudar você a criar um grande Software. Seguir as três etapas, na ordem em que elas foram propostas torna mais fácil o desenvolvimento de Softwares, pois elas visam que o mesmo deve satisfazer a todos: o cliente, pois o programa faz o que ele precisa; o programador, pois ele cria um projeto flexível e o seu chefe, pois as alterações ficam mais rápidas.

Você viu neste artigo como é simples segui-las e como princípios básicos da orientação a objetos fazem diferença quando o assunto é reutilização de código. Infelizmente não deu para mostrar um passo a passo de como aplicar a delegação de responsabilidade para comparar as especificações no método Pesquisar, mas no exemplo para download, você verá duas versões diferentes do Catálogo da Mary: uma com a delegação e outra sem a delegação. A diferença entre as duas versões é o método Pesquisar da classe TEstoque.

Agora é com você! Mary passou a vender bicicletas com suspensão dianteira. Baixe o download do artigo e use a versão com a delegação para fazer a alteração que Mary pediu – não esqueça que ela continua vendendo bicicletas sem suspensão. Uma dica: você vai precisar apenas de um novo atributo na classe TBicicletaEspec. Depois, você pode fazer a mesma alteração na versão sem a delegação para tirar suas próprias conclusões.

Rafael Stavarengo
faelsta@gmail.com
http://www.devmedia.com.br/stavarengo
Programador-analista da Cheina Informática. Trabalha com Delphi, JME, JSE e PHP + ZendFramework. Formando em Técnologia de Análise e Desenvolvimento de Sistemas pela UNIPAR.

Site do autor
http://www.devmedia.com.br/stavarengo

UML
http://pt.wikipedia.org/wiki/UML

Site do OMG
http://www.uml.org/

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Cursos Online

A **Revista WebMobile** oferece para seus assinantes uma série de Cursos Online de alto padrão de qualidade **.Conheça abaixo os cursos já disponíveis.**

Curso de .net em destaque

Aplicação completa de orçamento Doméstico no Visual Studio 2005

Confira neste curso online como criar uma aplicação completa no Visual Studio 2005, usando ASP.NET, Web Services, Mobile e muito mais! Veja como criar uma aplicação de orçamento doméstico, usando diagrama de classes de uma forma muita produtiva, criar classes de negócios de acesso a dados.

Confira o plano de aula completo:
www.devmedia.com.br/domesticovs

Curso de Java em destaque

Introdução ao desenvolvimento para celulares com J2ME

Confira neste curso os principais recursos do J2ME. Aprenda também o passo a passo para criar sua primeira aplicação J2ME. Neste curso você irá aprender diversas funcionalidades desta tecnologia para desenvolvimento de dispositivos móveis.

Confira o plano de aula completo:
www.devmedia.com.br/celularesj2me

Assine a **WebMobile** e comece já seu treinamento!
www.devmedia.com.br/assine

A sua melhor opção de aprendizagem!

Em breve mais novidades para você! www.devmedia.com.br/curso

DevMedia group Mais Informações: www.devmedia.com.br/central - Tel.: 21 3382-5038

Design Patterns -

Abstract Factory e Factory Methods

Quando precisamos flexibilizar nossas aplicações criamos uma complexa hierarquia de objetos, com vários níveis de descendência e várias classes irmãs com o mesmo ancestral em comum. Ao longo de nosso sistema precisamos passar objetos como parâmetros para métodos de outros objetos, e fazemos várias operações onde as instâncias configuram-se mutuamente.

Alguns impasses podem surgir como diminuir o acoplamento e a complexidade de uma aplicação, isolar a criação e configuração de um objeto da classe que precisará dele, sem alterar sua interface ou outros módulos que já dependem dele.

Vários cenários dão margem a esse impasses, como por exemplo:

- Trocar uma classe ou Unit sem fazer alterações que comprometam o funcionamento do sistema;
- Trocar o comportamento ou funcionalidades de nosso sistema apenas por adicionar ou trocar arquivos ou configurações;
- Instanciar classes e Forms, e executar seus métodos através de Strings;
- Criar famílias paralelas de objetos, com dados e operações diferentes, que podem ser facilmente trocados entre si pelo desenvolvedor ou até mesmo pelo usuário;
- Criar aplicações que funcionem com múltiplos bancos de dados e múltiplas tecnologias de acesso.

Dois padrões de projeto que tornam isso possível são o Factory Method e o Abstract Factory. Factory Method é um método de uma classe chamada Factory (fábrica) que cria e retorna um objeto de outra classe chamado Product, separando assim a lógica da criação de um objeto do seu uso. Ele permite também que as subclasses da fábrica definam qual o tipo correto de objeto a ser criado. Já a Abstract Factory pode ser uma interface comum ou uma classe ancestral de todas as fábricas. Ela tem os métodos abstratos para criação de produtos, mas delega às suas descendentes a inteligência de saber qual família de produtos criar.

Neste artigo iremos abordar os temas Factory Method e Abstract Factory com exemplos teóricos e práticos. Além dos exemplos, mostrar como essas técnicas podem auxiliar no desacoplamento entre as partes do seu sistema e algumas dicas para tornar isso mais viável.

Mostraremos como isolar nossas classes de negócio das classes de persistência, mantendo, porém uma interface comum

Resumo DevMan

De que trata o artigo:

Este artigo abordará o tema Design Patterns ou padrões de projeto. Explicaremos como funcionam os padrões de projeto criacionais Factory Method e Abstract Factory e demonstraremos como eles podem ser usados para criar uma aplicação com o menor nível de acoplamento possível entre os módulos. Serão mostrados vários exemplos de criação e utilização desses padrões com dificuldade progressiva. Como instanciar Forms e utilizá-las através de arquivos de configuração ou registros em banco de dados e como criar uma aplicação totalmente independente de banco de dados e de tecnologia de acesso.

Para que serve:

Os padrões de criação servem, principalmente, para isolar e desacoplar unidades com classes que precisam de um objeto das unidades que criam o objeto propriamente dito. O principal objetivo dos padrões mencionados é separar o processo de criação de um objeto de sua implementação. Ou seja, todas as units que usariam um determinado objeto não precisam conter uma referência à Unit que possui a declaração e implementação do mesmo.

Em que situação o tema é útil:

Aliada com o uso de interfaces, esta técnica favorece a criação e o uso de objetos e famílias inteiras de objetos que possam ser intercambiáveis, inclusive em tempo de execução. Essencial quando uma classe precisa de um objeto de uma certa interface, mas não sabe exatamente qual objeto instanciar, ou quando queremos deixar a criação de um objeto ou exibição de um formulário a cargo do usuário, onde ele pode decidir e configurar qual classe será usada. Aplicações flexíveis e com mínimo acoplamento devem usar esta técnica.

Design Patterns:

Cada vez mais nos é exigido que nossas aplicações sejam flexíveis e que possam ser usadas com vários bancos de dados. O custo, tempo e dificuldade de se fazer uma alteração desse tipo numa aplicação 100% RAD podem ser altíssimos. Mas ninguém transforma uma aplicação RAD já robusta e madura no mercado por algo totalmente Orientado a Objetos. Esse artigo mostrará técnicas básicas de padrões de projetos que podem ser usadas mesmo em aplicações RAD com componentes Data-Aware, pois o foco desta técnica é criar interfaces e objetos que possam ser configurados ou trocados em tempo de execução de maneira transparente para o usuário e fácil para o desenvolvedor, através de métodos de outras classes responsáveis por criar e configurar os objetos corretos. O principal exemplo disso seria a criação dos objetos de acesso a dados, dependendo do banco utilizado.

Parte 1

Vitor Luiz Rubio

entre elas, para que os dois lados da aplicação possam ser alterados livremente.

Para isso usaremos os padrões de criação Factory Method e Abstract Factory, mas antes precisaremos explicar esses conceitos.

Cliente: Podemos usar a palavra cliente para designar um objeto que precisa ou depende de outro. Por exemplo, se um ClientDataset depende de um DatasetProvider e que este esteja instanciado, o ClientDataset é o cliente, e o DatasetProvider o provedor. No caso dessas classes em especial o nome delas já diz tudo. Porém veremos mais adiante casos especiais de clientes e provedores.

Padrões de criação: São padrões de projeto usados para se construir classes ou métodos que criem e configurem objetos de outras classes. Eles podem ser usados para se criar várias formas de se instanciar um objeto, ou vários objetos ligados entre si, sem que o cliente (objeto utilizador) saiba como esses outros objetos são ligados entre si e sem que a Unit cliente precise ter uma referência a Unit do provedor. Estudaremos dois desses padrões, o Factory Method e o Abstract Factory, porém existem outros que não serão abordados nesse artigo, como Builder e Flyweight, só para citar dois.

Factory Method: É um padrão de projeto de Software que fornece uma interface para criação de famílias de objetos relacionados ou dependentes, sem especificar suas classes concretas. É um método de uma classe que cria um objeto. E Factory é uma classe que possui um ou mais factory methods. Esse padrão de projeto permite que a criação do objeto seja delegada para suas subclasses. Assim, se um cliente precisa de um objeto que faz parte de uma família ou série de classes que tem um mesmo ancestral em comum ou implementam a mesma interface mas não sabe qual classe necessariamente ele deve instanciar, ele simplesmente pode chamar um método fábrica de uma classe fábrica. Esse método será o responsável por saber que tipo de objeto criar e como inicializá-lo. Assim, a Unit do cliente não precisará ter em seu "uses" as Units dos provedores.

Abstract Factory: É um padrão de projeto que permite a criação de famílias de objetos relacionados ou dependentes, através de uma única interface e sem que a classe concreta seja especificada. Abstract Factory é uma classe abstrata mãe de todas as classes Factory, que têm a interface padrão de todas elas.

Um fato digno de nota é que toda classe já é uma Factory degenerada, porque possui o método "create" que cria um objeto da própria classe. Mesmo no Delphi Prism, a palavra chave "new" acompanhada de uma classe pode ser considerado um método fábrica de um objeto só.

Algumas regrinhas que vão facilitar a sua vida na hora de usar Factory Methods e Abstract Factory. Essas regras também vão lhe auxiliar a escrever um código com acoplamento mais baixo e menor interdependência entre as Units:

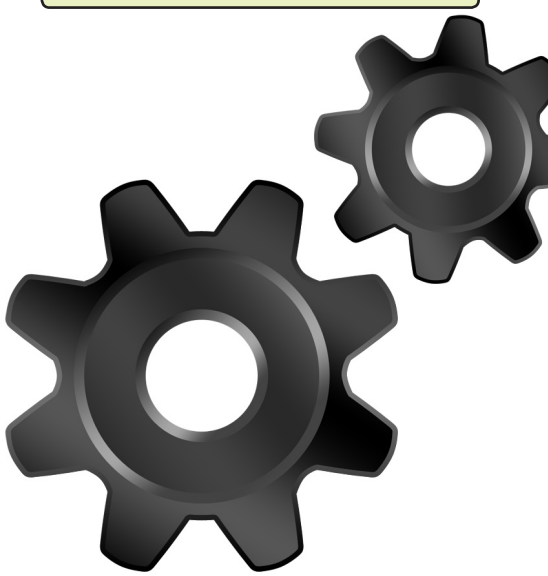
- Programe orientado a Interfaces: mais do que um contrato ou uma maneira de simular herança múltipla, as Interfaces são uma maneira de criar um grau de parentesco entre classes que não são parentes, nem mesmo se conhecem, e o que é melhor, intercambiar objetos dessas classes. Além disso, programar orientado a interfaces te força a implementá-las e ajuda a garantir que alterações em Units e classes continuem compactáveis contanto que continuem seguindo a risca as interfaces;
- Coloque uma classe por Unit, a menos que essas classes sejam tão correlacionadas que somente na mesma Unit seriam usadas no mesmo contexto;
- Coloque as Interfaces ou as classes abstratas, quando aplicável, em Units separadas;
- Quando criar um Form, não deixar que o Delphi a crie automaticamente na inicialização da aplicação, retirar do .dpr o "Application.CreateForm(TMyForm, MyForm);" e retirar até a variável publica global do Form criado na Unit do mesmo, afinal vamos trabalhar estritamente com POO;

Nota do DevMan

O Builder é um padrão de projeto que permite a separação da criação de um objeto da sua representação, de forma que este permita a construção de diferentes representações. Este padrão é muitas vezes comparado com ao Abstract Factory pois ambos podem ser utilizados para a construção de objetos complexos. A grande diferença entre ambos é que o Builder constrói apenas objetos complexos enquanto o Abstract Factory constrói famílias de objetos, simples ou complexos, de uma só vez.

Nota do DevMan

O padrão Flyweight é apropriado quando vários objetos devem ser manipulados, e esses não suportam dados adicionais. Neste não existem ponteiros para os métodos, pois isto consome muita memória. Em contrapartida são chamadas sub-rotinas diretamente para acessar as informações.



• Na hora de sobrecarregar um método de uma classe descendente, nunca se esqueça de colocar a palavra-chave **override** na declaração do método, para que o método da classe filha seja executado mesmo que seu objeto esteja sendo referenciado através de uma variável da classe mãe. Na classe mãe não se esqueça de marcar os métodos como **virtual** ou **dynamic** se você quiser que eles sejam substituídos pelos métodos filhos sobrecarregados. Isso é um princípio básico do Polimorfismo.

Para entender o Factory Method, temos um padrão que você pode observar na **Figura 1**.

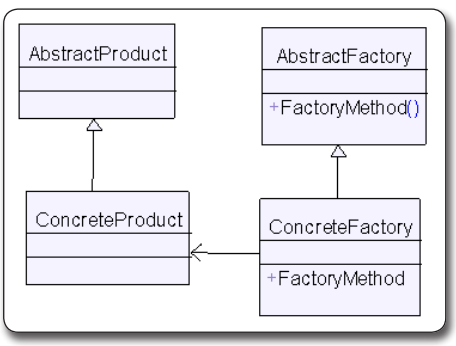


Figura 1. Diagrama de classes UML mostrando o Factory Method

Uma Concrete Factory é uma classe responsável por criar para nós um objeto de outra classe. Ela pode herdar de uma Abstract Factory se tivermos uma família muito grande de classes relacionadas, ou até mesmo muitas famílias. Mas pode também ser simplesmente uma classe independente responsável por criar a classe certa na hora certa.

Iniciando nosso exemplo

Para todos os nossos exemplos, vamos usar um Form com um botão. Este botão cria uma Factory e executa um Factory Method qualquer para criar um objeto. Será executado um método deste objeto e o resultado desse método será mostrado em um Memo.

Crie um novo projeto VCL Forms Application Win32 e salve o projeto com o nome padrão de "Project1.dpr" mesmo. O formulário padrão e a Unit padrão também podem ser salvos com o nome de "Unit1.pas".

Listagem 1. Units uAbstractProduct

```
//Unit uAbstractProduct
unit uAbstractProduct;

interface

type
 TAbstractProduct = class
 public
   function ShowWhoIAm: string; virtual; abstract;
 end;

implementation

{ TAbstractProduct }

end.
```

Listagem 2. Unit uConcreteProduct

```
//Unit uConcreteProduct
unit uConcreteProduct;

interface

uses
 uAbstractProduct;

type
 TConcreteProduct = class(TAbstractProduct)
 public
   function ShowWhoIAm: string; override;
 end;

implementation

{ TConcreteProduct }

function TConcreteProduct.ShowWhoIAm: string;
begin
 result := 'I am a ' + self.ClassName + ' object';
end;

end.
```

Crie uma nova Unit com o nome uAbstract product. Esta Unit terá uma classe abstrata chamada TAbstractProduct com um método abstrato chamado ShowWhoIAm que será sobrecarregado nas classes descendentes desta. A unit uAbstractProduct pode ser vista na **Listagem 1**.

Na unit uAbstractProduct criamos apenas uma classe abstrata chamada TAbstractProduct. Olhando no diagrama da **Figura 1** podemos ver que essa classe TAbstractProduct será a classe ancestral de todos os nossos produtos concretos.

Esta classe possui um método chamado ShowWhoIAm marcado como virtual e abstract. Foi marcado como virtual para que possa ser sobrecarregado dinamicamente e que objetos instanciados referenciados por variáveis do tipo abstrato executem o método correto, que é o método do descendente. O mesmo foi marcado como abstract porque a classe é abstrata, sendo a ancestral de todas as classes desse tipo, por isso não necessitará de implementação dos métodos.

Vamos criar nosso primeiro produto concreto, para isso, crie uma outra Unit chamada uConcreteProduct e coloque no uses desta a uAbstractProduct. Nesta unit você criará a classe TConcreteProduct que será filha de TAbstractProduct e sobrecarregará (override) o método ShowWhoIAm. Veja como a Unit uConcreteProduct ficará na **Listagem 2**.

Essa classe TConcreteProduct é apenas uma das várias classes que podem descender de TAbstractProduct. A ideia aqui é imaginar que quando um cliente necessitar de qualquer tipo de descendente do produto abstrato, mas não souber necessariamente qual, um método fábrica, baseando-se em parâmetros de configuração ou dados provenientes de uma base de dados, irá criar o produto concreto ideal para atender o cliente, e sem que a Unit do cliente "conheça" a Unit da classe concreta.

Na implementação do método Show WhoIAm, coloque uma mensagem qualquer concatenada com o nome da própria classe.

Crie mais uma Unit chamada uConcreteFactory que será nossa classe fábrica. Nela crie a classe TConcreteFactory com o método CreateProduct. O método CreateProduct cria e retorna um objeto do tipo TConcreteProduct, para isso, a Unit uConcreteFactory deve ter a uConcreteProduct declarada em seu uses. A **Listagem 3** mostra como ficará o código desta Unit.

Podemos reparar aqui que apenas a fábrica concreta tem conhecimento do produto concreto. O nosso "cliente" será o TForm1, o Form principal desse primeiro projeto, na unit1. Ela não precisará ter as Units dos produtos concretos em seu uses. Precisará apenas conhecer o ancestral de todos eles.

A Unit1 deve conhecer as units uConcrete Factory e uAbstractProduct. Para isso declare-as no seu uses. Crie no Form principal um Botão (button1) e um Memo (memo1). O Button1 cria uma instância de TConcreteFactory, usa o método Create-Product para criar um TConcreteProduct e armazena esse resultado numa variável do tipo TAbstractProduct, que é a classe mãe que fornece a Interface básica para todos os possíveis Concrete Products que você criar.A **Listagem 4** mostra a unit1 e o evento OnClick do Button1.

Como a única operação que fazemos com essa classe é mostrar o retorno do método ShowWhoIAm no Memo1, podemos destruir logo depois, com o método Free. O Form principal ficará como na **Figura 2**.

Qual é a vantagem disso tudo? Observe: a minha unit1 não possui a uConcreteProduct no seu uses e nem conhece a classe TConcreteProduct. Mesmo assim ela é capaz de usá-lo através da TConcreteFactory. Com isso eu posso trocar a implementação de TConcreteProduct, ou até trocar a classe inteira por qualquer outra que implemente a mesma interface ou também seja descendente de TAbstractProduct. Posso trocar esses Concrete Products à vontade, sem causar impacto no sistema, bastando modificar a minha TConcreteFactory. Posso usar a TConcreteFactory para decidir, através de parâmetros, através de arquivos .ini, ou através de configurações numa tabela de banco de dados qual tipo de Produto ela irá criar.

Listagem 3. Primeira fábrica concreta

```
//Unit uConcreteFactory
unit uConcreteFactory;

interface

uses uConcreteProduct;

type

 TConcreteFactory = class
 public
  function CreateProduct:TConcreteProduct;
 end;

implementation

{ TConcreteFactory }

function TConcreteFactory.CreateProduct:
 TConcreteProduct;
begin
 result := TConcreteProduct.Create;
end;

end.
```

Listagem 4. Unit Principal

```
unit Unit1;

interface

uses
 Windows, Messages, SysUtils, Variants, Classes,
 Graphics, Controls, Forms,
 Dialogs, uAbstractProduct, uConcreteFactory,
 StdCtrls;

type
 TForm1 = class(TForm)
  Button1: TButton;
  Memo1: TMemo;
  procedure Button1Click(Sender: TObject);
 private
  { Private declarations }
 public
  { Public declarations }
 end;

var
 Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
var
 aProduct: TAbstractProduct;
begin
 with TConcreteFactory.Create do
 begin
  aProduct := CreateProduct;
  Free;
 end;
 Memo1.Lines.Append( aProduct.ShowWhoIAm );
 aProduct.Free;
end;

end.
```

Adicionando mais um objeto concreto

Crie em seu projeto mais uma unit chamada uAnotherConcreteProduct. Coloque no uses da seção interface a Unit uAbstractProduct. Crie outra classe filha de TAbstractProduct chamada TAnother-ConcreteProduct. Sobrecarregue o método ShowWhoIAm para mostrar uma outra mensagem. A **Listagem 5** contém o código dessa unit. Nada de muito especial, ela apenas ilustra como uma segunda classe concreta pode ser totalmente diferente da primeira. No nosso caso as duas são "irmãs" e descendem da mesma mãe.

Criamos uma unit simples com um outro tipo de produto concreto. A diferença aqui está na mensagem, mas poderia ser na implementação de qualquer método. O importante é que ele é irmão do TConcreteProduct, pois ambos descendem de TAbstractProduct. Repare aqui que dependendo de como isso é implementado, uma Interface comum pode ser usada no lugar de uma classe abstrata ancestral.

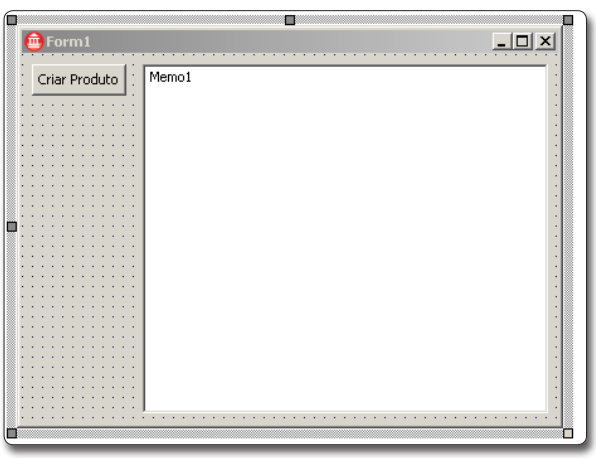


Figura 2. Form1 do primeiro exemplo

Agora, no seu Form1, como saber qual das duas classes a minha Concrete Factory vai criar? Como o próprio livro "Padrões de Projeto" sugere, isso é flexível. São admitidas algumas variantes desse padrão. Por exemplo, sua Factory pode ter dois Factory Methods: um CreateProduct e um CreateAnotherProduct. Ou simplesmente

Nota

Note que aqui usamos uma classe abstrata para nosso TAbstractProduct, mas poderíamos usar uma interface IAbstract-Product, sendo que os produtos concretos poderiam implementá-la. Isto daria bastante flexibilidade ao projeto.

o método CreateProduct pode aceitar um parâmetro ou pode consultar um arquivo .ini para saber qual o produto correto. Vamos ver os três métodos: Abra a Unit u ConcreteFactory e coloque a uAnother-ConcreteProduct no uses. Sempre a fábrica concreta deve ter uma referência às unidades dos produtos concretos, assim, além de saber como criá-los, ele elimina a necessidade do cliente ter referências a todos os objetos concretos. Inclua o método CreateAnotherProduct conforme a **Listagem 6**.

O importante aqui é sempre lembrar que o padrão Factory isola a criação de um objeto de sua lógica. Isso quer dizer que objetos com muitos parâmetros de criação, com muitos atributos a serem configurados, ou objetos que precisam estar associados com vários outros objetos menores para funcionarem como um todo, podem ser pré-criados e pré-relacionados por um método fábrica, permitindo assim a entrega do objeto pronto para uso ao cliente e poupando o cliente de ter que saber a "receita" para criar o objeto.

Chame agora os dois métodos do seu Concrete Factory no formulário principal. Você pode colocar dois botões, perguntar para o usuário com um InputBox, criar um arquivo ini, ou o que você preferir. Observe na **Listagem 7** como modificamos o botão do formulário principal.

Esse tipo de abordagem que usamos tem dois inconvenientes: Primeiro precisamos criar mais métodos na nossa Concrete Factory, o que modifica sua Interface. E segundo precisamos colocar na classe cliente a responsabilidade de decidir qual produto criar.

Você pode flexibilizar um pouco mais fazendo com que a sua ConcreteFactory, ao invés de ter vários métodos, aceite vários parâmetros. Para isso, o método chamado deve retornar um TAbstractProduct, mesmo que dentro dele seja criado um TConcreteProduct. Se isso fosse feito desde o começo já flexibilizaria nosso código. Para isso, abra a Unit uConcreteFactory e adicione no uses a uAbstractProduct. A **Listagem 8** mostra como o projeto pode ganhar um pouco mais de flexibilidade usando como retorno do método fábrica do tipo de pro-

Listagem 5. Unit com outro produto concreto

```
unit uAnotherConcreteProduct;

interface

uses
 uAbstractProduct;

type
 TAnotherConcreteProduct = class(TAbstractProduct)
 public
  function ShowWhoIAm: string; override;
 end;

implementation

{ TAnotherConcreteProduct }

function TAnotherConcreteProduct.ShowWhoIAm: string;
begin
 result := 'Dependendo da situação eu posso ser um ' +
 self.ClassName + '!';
end;

end.
```

Listagem 6. Novo método para criação do novo tipo de objeto

```
unit uConcreteFactory;

interface

uses
 uConcreteProduct, uAnotherConcreteProduct;

type

 TConcreteFactory = class
 public
  function     CreateProduct:TConcreteProduct;
  function CreateAnotherProduct: TAnother
   ConcreteProduct;
 end;

implementation

{ TConcreteFactory }

function TConcreteFactory.CreateAnotherProduct:
 TAnotherConcreteProduct;
begin
 result := TAnotherConcreteProduct.Create;
end;

function TConcreteFactory.CreateProduct:
 TConcreteProduct;
begin
 result := TConcreteProduct.Create;
end;

end.
```

Listagem 7. Invocando os dois métodos fábrica

```
procedure TForm1.Button1Click(Sender: TObject);
var
 aProduct, aAnotherProduct: TAbstractProduct;
begin
 with TConcreteFactory.Create do
 begin
  aProduct := CreateProduct;
  aAnotherProduct := CreateAnotherProduct;
  free;
 end;

 Memo1.Lines.Append( aProduct.ShowWhoIAm );
 Memo1.Lines.Append( aAnotherProduct.
 ShowWhoIAm );

 aProduct.Free;
 aAnotherProduct.Free;
end;
```

Listagem 8. Novo método fábrica

```
unit uConcreteFactory;

interface

uses
 uAbstractProduct, uConcreteProduct, uAnotherCon-
creteProduct;

type
 TConcreteFactory = class
 public
  function CreateProduct(TypeProduct: string):
   TAbstractProduct;
 end;

implementation

{ TConcreteFactory }

function TConcreteFactory.
CreateProduct(TypeProduct: string):
 TAbstractProduct;
begin
 if TypeProduct = 'ConcreteProduct' then
  result := TConcreteProduct.Create
 else if TypeProduct = 'AnotherConcreteProduct' then
  result := TAnotherConcreteProduct.Create
 else
  Result := nil;
end;

end.
```

duto abstrato, mas internamente usando algum algoritmo para criar o tipo correto de produto concreto. Como parâmetro para o método usaremos uma String, mas você pode usar um integer ou enum.

Na **Listagem 8**, caso o valor do argumento TypeProduct seja inesperado estamos retornando nil. Você pode também disparar uma exception. O importante é que tenhamos o controle do argumento, se é inválido, e também nos permita saber se realmente um objeto foi criado ou não.

Neste exemplo o que mudou foi o método fábrica. Mas a responsabilidade de saber qual objeto criar ainda está no Form principal, apenas abstraímos esse conhecimento através de Strings. Você pode fazer com que as Strings venham de bancos de dados ou de arquivos .ini. A **Listagem 9** mostra como fica nosso formulário principal criando os objetos através de Strings.

Esses poucos passos que fizemos já ilustraram basicamente o que são Factory Methods e como podemos usá-los para flexibilizar nossas aplicações. Faremos agora um outro teste transferindo para a nossa Concrete Factory a responsabilidade de saber qual tipo de objeto criar. Para isso usaremos um arquivo ini. Não teremos mais que passar o tipo de objeto como parâmetro para nosso Factory Method.

Abra a sua uConcreteFactory e coloque no uses as Units IniFiles e Forms. Precisamos da Unit Forms para acessar o objeto Application. Nosso arquivo .ini terá o nome conforme: <NomeDaAplicação>.exe.ini. Abra o bloco de notas e crie o arquivo Project1.exe.ini, adicionando as linhas a seguir:

```
[Aplicacao]
TipoProduto=ConcreteProduct
```

Volte o código da unit1 para ficar da maneira como estava no nosso primeiro exemplo, como é mostrado na **Listagem 10**.

Dessa maneira o método concentrará toda a inteligência para criar produtos concretos sem que precisemos nem passar o tipo como parâmetro. Altere a uConcreteFactory para que fique como no código da **Listagem 11**. Isso fará com que o tipo de produto correto venha do arquivo .ini criado anteriormente.

Listagem 9. Criando dois objetos diferentes com o mesmo método

```
procedure TForm1.Button1Click(Sender: TObject);
var
 aProduct, aAnotherProduct: TAbstractProduct;
begin
 with TConcreteFactory.Create do
 begin
  aProduct := CreateProduct('ConcreteProduct');
  aAnotherProduct := CreateProduct('Another
  ConcreteProduct');
  Free;
 end;
 Memo1.Lines.Append( aProduct.ShowWhoIAm );
 Memo1.Lines.Append( aAnotherProduct.Show
 WhoIAm );
 aProduct.Free;
 aAnotherProduct.Free;
end;
```

Listagem 10. Um único método fábrica, sem parâmetros

```
procedure TForm1.Button1Click(Sender: TObject);
var
 aProduct: TAbstractProduct;
begin
 with TConcreteFactory.Create do
 begin
  aProduct := CreateProduct;
  Free;
 end;
 Memo1.Lines.Append( aProduct.ShowWhoIAm );
 aProduct.Free;
end;
```

Listagem 11. Método fábrica com arquivo INI

```
function TConcreteFactory.CreateProduct:
TAbstractProduct;
var
 TypeProduct: string;
begin
 with TIniFile.Create(Application.ExeName+'.ini') do
 try
  TypeProduct := ReadString('Aplicacao','TipoProduto','');
  if TypeProduct = 'ConcreteProduct' then
   result := TConcreteProduct.Create
  else if TypeProduct = 'AnotherConcreteProduct' then
   result := TAnotherConcreteProduct.Create
  else
   Result := nil;
 finally
  free;
 end;
end;
```

O que estamos fazendo aqui é usar Application.Exename (por isso precisamos da Unit Forms) para obter o nome da aplicação. Isso porque criamos e utilizamos o arquivo .ini com o mesmo nome da aplicação concatenado com a extensão .ini, ficando project1.exe.ini.

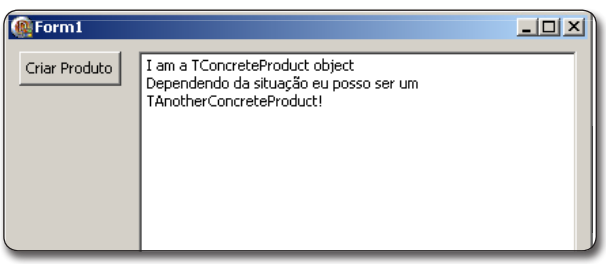


Figura 3. Os dois produtos, depois de se alterar o arquivo .ini

Acessamos a seção "Aplicacao", chave "TipoProduto" através do método "ReadString" do TIniFile. O restante do método é igual ao último exemplo. Repare que embora agora o método não tenha parâmetros, estamos utilizando a String TypeProduct como variável local.

Teste a aplicação, depois abra o arquivo .ini e altere a variável TipoProduto para AnotherConcreteProduct e teste novamente. O resultado pode ser visto na **Figura 3**.

Veja que retiramos várias responsabilidades do nosso Form, que precisa usar os produtos concretos, mas não sabe quais. Mas nosso exemplo está bastante teórico. Vamos tornar esse exemplo mais prático usando o seguinte cenário: Um ERP que pode ser customizado para atacado e varejo. Caso o sistema seja configurado para atacado ele terá como objeto cliente sempre pessoas jurídicas, o Form de cadastro de clientes será específico para pessoas jurídicas e haverá nota fiscal. Já no caso do varejo, o cliente é do tipo pessoa física, o formulário de cadastro é específico para esse tipo de cliente e terá cupom fiscal no lugar de nota fiscal. O formulário principal, que tem os métodos para criar os outros Forms, não deve conhecer a existência de nenhuma dessas outras classes.

Um exemplo mais prático

Vamos aproveitar esse exemplo pronto, mas excluiremos as duas units que têm os

produtos concretos, excluindo-as também do projeto. Adicione um Form no projeto chamado de FrmCadastroAbstrato. Ele será a base para nossos formulários de cadastro (Essa é uma ótima oportunidade de usarmos Interfaces. Veremos como mais adiante).

Coloque nesse Form um método virtual abstrato CarregarCliente que aceite como parâmetro uma variável do tipo TAbstract Product. O formulário base é mostrado na **Figura 4** e seu código é mostrado na **Listagem 12.**

Criamos um formulário em branco, com um único método chamado CarregaCliente. Esse método será abstrato, significando que será implementado nos descendentes de TFrmCadastroAbstrato. O método aceitará como parâmetro um objeto TAbstractProduct que na verdade será qualquer um de seus descendentes.

Criaremos mais duas Units, uma chamada uClientePF com a classe TPessoaFisica e a outra chamaremos uClientePJ e criaremos a classe TPessoaJuridica. Tanto a classe para pessoa física como a classe para pessoa jurídica serão descendentes de TAbstractProduct e possuirão o método ShowWhoIAm. As **Listagens 13** e **14** mostram cada uma dessas classes. São classes bem simples, sua implementação não tem segredo.

Cada uma dessas classes terá uma propriedade referente ao seu nome e documento, além do método ShowWhoIAm, que será uma concatenação de suas propriedades.

Adicione outros dois Forms descendentes do FrmCadastroAbstrato, o FrmCadastroPF e o FrmCadastroPJ. No FrmCadastroPF coloque dois Edits rotulados como Nome e CPF. No FrmCadastroPJ coloque dois Edits rotulados como Razão Social e CNPJ. Nesses dois Forms você irá sobrecarregar o método CarregarCliente. Você fará o Type Cast para o tipo de objeto correto dentro desse método, e não precisa se preocupar se o objeto passado como parâmetro será mesmo do tipo correto porque se a ConcreteFactory estiver implementada corretamente, ela garantirá que os objetos criados serão da mesma família e compatíveis entre si.

A **Figura 5** mostra como ficará o formulário de cadastro de pessoa física, seu código pode ser visto na **Listagem 15**. Já na **Figura 6** veremos a tela do cadastro de pessoa jurídica, bem como sua codificação na **Listagem 16**.

Observe aqui nas **Listagens 15** e **16** que ainda não temos uma situação ideal. No nosso exemplo os formulários de cadastro precisam ter um "conhecimento" das classes concretas TPessoaFisica e TPessoaJuridica.

Isso acontece porque propositalmente criamos os nomes das propriedades diferentes. Há três saídas para

Listagem 12. Formulário TFrmCadastroAbstrato

```
unit uFrmCadastroAbstrato;

interface

uses
 Windows, Messages, SysUtils, Variants, Classes,
  Graphics, Controls, Forms,
 Dialogs, uAbstractProduct;

type
 TFrmCadastroAbstrato = class(TForm)
 private
  { Private declarations }
 public
  { Public declarations }
  procedure CarregaCliente(cliente: TAbstractProduct);
  virtual; abstract;
 end;


implementation

{$R *.dfm}

end.
```

Listagem 13. Classe TPessoaFisica

```
unit uTPessoaFisica;

interface

uses
 uAbstractProduct;

type
 TPessoaFisica = class(TAbstractProduct)
 private
  FNome: string;
  FCPF: string;
  function GetNome:string;
  function GetCPF:string;
  procedure SetNome(valor:string);
  procedure SetCPF(valor:string);
 public
  function ShowWhoIAm:string; override;
  property Nome: string read GetNome write
  SetNome;
  property CPF: string read GetCPF write SetCPF;
 end;

implementation

{ TPessoaFisica }

function TPessoaFisica.GetCPF: string;
begin
 result := FCPF;
end;

function TPessoaFisica.GetNome: string;
begin
 result := FNome;
end;

procedure TPessoaFisica.SetCPF(valor: string);
begin
 FCPF := valor;
end;

procedure TPessoaFisica.SetNome(valor: string);
begin
 FNome := valor;
end;

function TPessoaFisica.ShowWhoIAm: string;
begin
 result := 'Nome:' + FNome + ' CPF:' + FCPF;
end;

end.
```

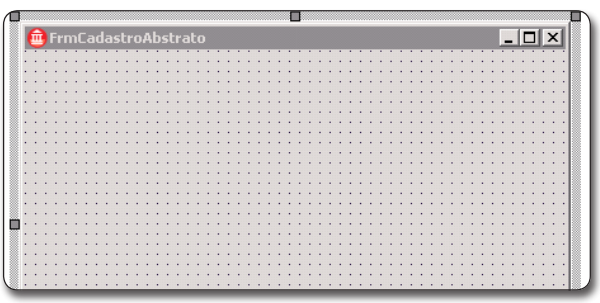


Figura 4. Formulário ancestral dos cadastros

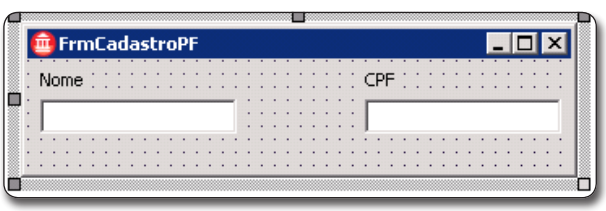


Figura 5. Cadastro de Pessoa Física

Nota

Para criar um formulário "filho" ou descendente de outro, clique em File>New>Other, e na pasta inheritable itens é possível visualizar os formulários de seu projeto. Basta escolher qual você deseja e clicar em OK que será criado um "herdeiro".

essa situação: ou colocar as propriedades com o mesmo nome nas duas classes, sendo que a classe mãe possua ambas as propriedades a segunda, que seria fazer com que as duas classes implementem uma mesma interface e a terceira seria usar RTTI para "descobrir" quais as propriedades que a classe tem e alterar seus valores. Nesses três casos os formulários concretos precisariam conhecer apenas o produto abstrato, ou a Interface comum entre os produtos concretos.

Edite o arquivo .ini com o bloco de notas e insira o texto a seguir:

```
[Aplicacao]
TipoEmpresa=Atacado
```

Com isso poderemos fazer a configuração entre atacado e varejo.

Nosso formulário principal não precisará mais do Memo, podemos deletá-lo. Também teremos de abrir a Unit uConcreteFactory e retirar do uses as duas Units que deletamos.

Ainda na Unit uConcreteFactory adicione as 5 novas Units: os dois Forms e as pessoas física e jurídica, além do Form de cadastro abstrato. Adicione também um método CreateInterface que retorne um TFrmCadastroAbstrato. Veja como ficará o código na **Listagem 17**.

O que fizemos foi construir uma fábrica que além de criar o tipo de pessoa correto, criará o tipo de formulário correto para lidar com ela. Estamos falando de duas famílias de objetos concretos distintas: a família dos "formulários" e a família das "pessoas". Em uma "tacada" só podem ser criados um objeto Pessoa e o formulário correto para lidar com ele.

Também estamos preenchendo alguns dados a título de teste e por questões de simplicidade, porém, não é função do método fábrica preencher esses dados. Mesmo assim isso mostra como o objeto criado pode ser devolvido já com algumas propriedades configuradas.

Listagem 14. Classe TPessoaJuridica

```
unit uTPessoaJuridica;

interface

uses
 uAbstractProduct;

type
 TPessoaJuridica = class(TAbstractProduct)
 private
  FRazao: string;
  FCNPJ: string;
  function GetRazao:string;
  function GetCNPJ:string;
  procedure SetRazao(valor:string);
  procedure SetCNPJ(valor:string);
 public
  function ShowWhoIAm: string; override;
  property Razao: string read GetRazao write SetRazao;
  property CNPJ: string read GetCNPJ write SetCNPJ;
 end;

implementation

{ TPessoaJuridica }

function TPessoaJuridica.GetCNPJ: string;
begin
 result := FCNPJ;
end;

function TPessoaJuridica.GetRazao: string;
begin
 result := FRazao;
end;

procedure TPessoaJuridica.SetCNPJ(valor: string);
begin
 FCNPJ := valor;
end;

procedure TPessoaJuridica.SetRazao(valor: string);
begin
 FRazao := valor;
end;

function TPessoaJuridica.ShowWhoIAm: string;
begin
 result := 'Razão: ' + FRazao + '  CNPJ: ' + FCNPJ;
end;

end.
```

Listagem 15. Código do formulário de pessoa física

```
unit uFrmCadastroPF;

interface

uses
 Windows, Messages,
 SysUtils, Variants,
 Classes, Graphics,
 Controls, Forms, Dialogs,
 uFrmCadastroAbstrato, uAbstractProduct,
 uTPessoaFisica, StdCtrls;

type
 TFrmCadastroPF = class(TFrmCadastroAbstrato)
  edNome: TEdit;
  edCpf: TEdit;
  Label1: TLabel;
  Label2: TLabel;
 private
  { Private declarations }
 public
  { Public declarations }
  procedure CarregaCliente(cliente: TAbstract
   Product); override;
 end;

implementation

{$R *.dfm}

{ TFrmCadastroPF }

procedure TFrmCadastroPF.CarregaCliente(cliente:
TAbstractProduct);
begin
 edNome.Text := (cliente as TPessoaFisica).Nome;
 edCpf.Text :=  (cliente as TPessoaFisica).CPF;
 self.Caption := cliente.ShowWhoIAm;
end;
end.
```

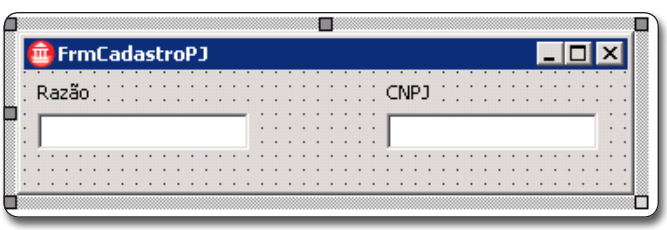


Figura 6. Cadastro de Pessoa Jurídica

Uma questão importante, para evitar Memory Leaks é que não estamos criando esses formulários com um owner, mas estamos deixando o mesmo como nil. Isso significa que a responsabilidade de destruir o formulário será sua. Você pode usar release, que destrói o formulário e limpa a fila de mensagens da API do Windows destinadas a ele.

Listagem 16. Código do formulário de pessoa jurídica

```
unit uFrmCadastroPJ;

interface

uses
 Windows, Messages, SysUtils, Variants, Classes,
 Graphics, Controls, Forms,
 Dialogs, uFrmCadastroAbstrato, StdCtrls,
 uAbstractProduct, uTPessoaJuridica;

type
 TFrmCadastroPJ = class(TFrmCadastroAbstrato)
   Label1: TLabel;
   edRazao: TEdit;
   Label2: TLabel;
   edCnpj: TEdit;
 private
  { Private declarations }
 public
  { Public declarations }
 procedure CarregaCliente(cliente: TAbstractProduct);
   override;
 end;

implementation

{$R *.dfm}

{ TFrmCadastroPJ }

procedure TFrmCadastroPJ.CarregaCliente(cliente: TAbstractProduct);
begin
 edRazao.Text := (cliente as TPessoaJuridica).Razao;
 edCnpj.Text :=  (cliente as TPessoaJuridica).CNPJ;
 self.Caption := cliente.ShowWhoIAm;
end;

end.
```

Listagem 17. Código da fábrica concreta

```
unit uConcreteFactory;

interface

uses
 uAbstractProduct, inifiles, forms, uFrmCadastroAbstrato;

type

 TConcreteFactory = class
 public
   function CreateProduct:TAbstractProduct;
   function CreateInterface: TFrmCadastroAbstrato;
 end;

implementation

uses uFrmCadastroPF, uFrmCadastroPJ, uTPessoaFisica, uTPessoaJuridica;

{ TConcreteFactory }

function TConcreteFactory.CreateInterface: TFrmCadastroAbstrato;
var
 TypeProduct: string;
begin
 with TIniFile.Create(Application.ExeName+'.ini') do
 try
  TypeProduct := ReadString('Aplicacao', 'TipoEmpresa', '');
  if TypeProduct = 'Atacado' then
  begin
   //estamos criando o owner como nil, então não esqueça de liberar essa
    //memória com release.
   result := TFrmCadastroPJ.Create(nil);
  end
  else if TypeProduct = 'Varejo' then
  begin
   //estamos criando o owner como nil, então não esqueça de liberar essa
   //memória com release.
   result := TFrmCadastroPF.Create(nil);
  end
  else
   Result := nil;
 finally
   free;
 end;

end;

function TConcreteFactory.CreateProduct: TAbstractProduct;
var
 TypeProduct: string;
begin
 with TIniFile.Create(Application.ExeName+'.ini') do
 try
  TypeProduct := ReadString('Aplicacao', 'TipoEmpresa', '');
  if TypeProduct = 'Atacado' then
  begin
   result := TPessoaJuridica.Create;
   {
    não é tarefa da ConcreteFactory preencher esses dados. Estamos
    preenchendo-os para  fins de teste
   }
   (result as TPessoaJuridica).Razao := 'Rubisoft Software';
   (result as TPessoaJuridica).CNPJ := '11111111000111';
  end
  else if TypeProduct = 'Varejo' then
  begin
   result := TPessoaFisica.Create;
   {
    não é tarefa da ConcreteFactory preencher esses dados. Estamos
    preenchendo-os para  fins de teste
   }
   (result as TPessoaFisica).Nome := 'Vitor Luiz Rubio';
   (result as TPessoaFisica).CPF := '11111111111';
  end
  else
   Result := nil;
 finally
  free;
 end;
end;

end.
```

Listagem 18. Formulário principal

```
unit Unit1;

interface

uses
 Windows, Messages,
 SysUtils, Variants,
 Classes, Graphics,
 Controls, Forms,
 Dialogs, uAbstractProduct,
 uConcreteFactory, uFrmCadastroAbstrato, StdCtrls;

type
 TForm1 = class(TForm)
  Button1: TButton;
  procedure Button1Click(Sender: TObject);
 private
  { Private declarations }
 public
  { Public declarations }
 end;

var
 Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
var
 aProduct: TAbstractProduct;
 aGui: TFrmCadastroAbstrato;
begin
 with TConcreteFactory.Create do
 begin
  aProduct := CreateProduct;
  aGui := CreateInterface;
  free;
 end;

 with aGui do
 begin
  CarregaCliente(aProduct);
  ShowModal;
  //estamos chamando release porque é a melhor forma de liberar um form da
  //memória.
  //a form será liberada depois que todas as mensagens pendentes a ela do
  //windows forem repassadas, e então release chama o free.
  //você pode implementar o release no onclose
  Release;
  aProduct.Free;
 end;
end;

end.
```

O release pode ser chamado no evento OnClose do formulário ou após o mesmo ser usado. O Form principal irá criar um objeto do tipo pessoa (TAbstractProduct) e um objeto do tipo TFrmCadastroAbstrato. Caberá ao ConcreteFactory criar os tipos corretos de Pessoa (física ou jurídica) e Forms de cadastro. Coloque no uses do Form principal a Unit do nosso Form base: uFrmCadastroAbstrato.

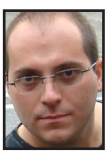
Lembre-se: a ConcreteFactory é quem criará tanto as interfaces com o usuário como os objetos de negócio e as classes DAO (de persistência) corretas e compactáveis. Ainda cabe à ConcreteFactory usar as classes DAO corretas para trazer o objeto correto do banco de dados.

Altere o formulário principal para que use a fábrica para gerar os objetos de que precisa, chamando os métodos correspondentes para criar a classe e para criar a GUI. A **Listagem 18** ilustra como ficará a unit1.

Conclusão

Faça o teste mudando a configuração do arquivo .ini de "Atacado" para "Varejo" e vice-versa e acompanhe o resultado. Verá que nossa Concrete Factory cria tanto o Form de cadastro de cliente como o próprio objeto cliente correto de acordo com a configuração. Como já dito, criamos um acoplamento desnecessário entre as Units dos Forms de cadastro concretas e seus produtos concretos correspondentes. Para corrigir isso podemos mudar a classe TAbstractProduct adicionando os métodos ou propriedades que necessitamos, ou mesmo usando uma interface no lugar da classe.

Vamos fazer isso no próximo artigo, onde criaremos mais um nível de desacoplamento, implementando uma Abstract Factory, para postergar às suas classes filhas concretas a responsabilidade de implementar os Factory Methods e produtos corretos. Até o próximo artigo.



Vitor Luiz Rubio
vitorrubio@gmail.com - http://www.vitorrubio.blogspot.com
Tecnólogo em Processamento de Dados pela FATEC-SP e MCP. Atua como Analista de Sistemas Sênior na Editora Revista dos Tribunais. Programa com Delphi desde a versão 3 e com .Net desde a versão 1.1.

Blogs do autor
http://www.vitorrubio.blogspot.com/
http://twitter.com/vitorrubio

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Delphi Prism e ASP

Reaproveite seus conhecimentos Win32 / De

Neste artigo vamos falar muito de ADO.NET, tanto os componentes do provider para SQL Server quanto da classe DataSet. Esta classe faz parte do grupo de componentes desconectados. Podemos comparar o DataSet ao ClientDataSet da VCL do Delphi Win32, pois permite armazenar dados em memória. Porém, de nada adianta colocarmos estes dados em memória se não persistirmos esses dados no servidor de alguma forma, visto que, após processar uma requisição, o ASP.NET destrói tudo da memória (devido a sua natureza state-less). Para isso, veremos como usar o objeto Session, que permite gerenciar uma sessão de usuário.

Um DataSet pode conter dados vindos de praticamente qualquer banco de dados, utilizando para isso uma "ponte" através de um Managed Provider, como para o SQL Server que estamos utilizando até aqui. Assim como o ClientDataSet, um DataSet independe de fonte dados, podendo conter dados vindos de arquivos XML, Web Services, criados somente em memória (estilo tabelas temporárias – o que faremos aqui) entre outros.

Na verdade, quando utilizamos o SqlDataSource até aqui, fizemos uso de um DataSet por baixo (dê uma olhada na propriedade DataSourceMode do controle). Como comentei anteriormente, o SqlDataSource não faz muita coisa além de ser uma "casca fina" em cima do ADO.NET. A partir deste artigo, nós veremos como utilizar ADO.NET no "braço", ou seja, vamos codificar. Além disso, também veremos como usar também conexões e DataReaders do ADO.NET (conectado), em vista disso, vale uma introdução à tecnologia.

Introdução ao ADO.NET

O ADO.NET é a tecnologia de acesso a dados do .NET Framework. Uma evolução do ADO (Active Data Objects), é totalmente escrito com código gerenciado (Managed Code). Uma importante mudança com relação ao ADO é que os DataSets são desconectados, por padrão – algo muito semelhante ao que temos no Delphi com a arquitetura dbExpress/DataSnap e o componente ClientDataSet. Ou seja, os dados são trazidos para uma cache local, a conexão é fechada, o cliente trabalha "off-line" e, quando precisa atualizar os dados, uma nova conexão é

Resumo DevMan

De que trata o artigo:

Lançado pela Embarcadero, o Delphi Prism é a nova forma de desenvolver aplicações para o .NET Framework e Mono utilizando o Object Pascal e o IDE do Visual Studio. Nesta série de artigos, veremos como utilizar o Delphi Prism para criar um site de E-Commerce usando a tecnologia ASP.NET.

Para que serve:

O objetivo principal desta série de artigos é familiarizar os desenvolvedores Delphi Win32 com o ambiente de desenvolvimento do Delphi Prism, o IDE do Visual Studio, bem como introduzir o desenvolvimento de aplicações Web. Dessa forma, faremos vários comparativos entre os dois IDEs e também entre os Frameworks ASP.NET e VCL, servindo como um excelente guia para diminuir a curva de aprendizado ao se aprender ASP.NET baseando-se nos seus prévios conhecimentos em Delphi Win32 (Desktop).

Em que situação o tema é útil:

Esta série será muito útil para quem nunca desenvolveu aplicativo algum para Web, nem mesmo teve um primeiro contato com o Delphi Prism / Visual Studio. Ao desenvolver aplicações para Web, ganhamos inúmeras vantagens se compararmos ao ambiente Desktop: Deploy simplificado, uso de computação nas nuvens, não há necessidade de instalações visto que os usuários acessam um mesmo servidor na internet, não há dependência do sistema operacional, pois usamos um browser, não há preocupação com backups nos clientes, nem altos custos para manutenções e visitas aos clientes etc.

Delphi Prism e ASP .NET:

No primeiro artigo desta série, tivemos uma introdução ao Delphi Prism, seus objetivos e propósitos. A seguir partimos para a criação do banco de dados no SQL Server usado ao longo desta série. Na segunda parte iniciamos a criação do Web Site, aproveitando para falar dos fundamentos do ASP.NET e Visual Studio, sempre comparando-os com a VCL e RAD Studio, com o intuito de facilitar o aprendizado. Vimos a estrutura básica de um Web Site e iniciamos a construção do layout básico do site de E-Commerce (chamado "Computer Store"), bem como o sistema de navegação. Na parte 3, vimos como construir a página principal do site, exibindo a lista de produtos, mostrando como fazer a conexão com o banco de dados. Na parte anterior vimos dois importantes passos do site, tarefas comuns realizadas em qualquer tipo de aplicação Web que trabalhe com banco de dados: pesquisas com parâmetros. Nesta parte da série, vamos adicionar uma das principais funcionalidades do site, a possibilidade do usuário adicionar produtos a um carrinho de compras.

Nota

Como já comentei anteriormente, existem excelentes frameworks de persistência e mapeamento objeto / relacional para .NET, como o NHibernate e ADO.NET Entity Framework. Estudaremos ambos em futuras oportunidades.

.NET
sktop para criar aplicações Web – Parte 5

Guinther Pauli

restabelecida. Isso torna aplicações do ADO.NET muito escaláveis permitindo que servidores atendam a uma demanda maior de conexões mantendo um desempenho consistente.

Algumas vezes, no entanto, não é uma boa solução conectar e reconectar a um banco de dados a cada transação de uma aplicação cliente. Para esses casos, o ADO.NET oferece um mecanismo de pool de conexões, com o qual uma mesma conexão pode permanecer ativa e atender a um grande número de solicitações.

O ADO.NET foi construído especialmente para aplicações multicamadas e para Web. Antes dele, o COM/COM+ era sinônimo de programação distribuída/multicamadas no Windows; no entanto utilizar COM em aplicações multicamadas dificulta a comunicação com clientes que rodem em outras plataformas, além de não ser uma boa solução quando se pretende distribuir informações na Web (devido a limitações impostas por Firewalls, por exemplo).

Diferentemente do ADO (que é baseado no COM), o ADO.NET utiliza XML para intercâmbio de dados ganhando, com isso, recursos de integração e interoperabilidade. Além disso, o ADO.NET pode ser utilizado em qualquer linguagem que dê suporte ao .NET, como VB.NET, C# e Delphi Prism. Isso significa que, se você conhecer o ADO.NET (e demais objetos do .NET Framework), poderá desenvolver aplicações em qualquer linguagem com relativa facilidade.

A seguir conheceremos os principais objetos que compõem a arquitetura ADO.NET. As classes do ADO.NET estão divididas em dois grandes grupos: Managed Providers (provedores gerenciados) e Content Components (componentes de conteúdo).

Managed Providers

Os componentes do grupo Managed são responsáveis pelo acesso a dados. Incluem classes para conexão ao banco, gerenciamento de transações, execução de comandos e leitura de dados. Atualmente são distribuídos alguns Managed Providers nativos com o .NET Framework, como SQL Server Data Provider, OLE DB Data Provider e ODBC. Você utiliza o primeiro tipo quando precisar criar uma aplicação que se comunique exclusivamente com SQL Server, é o nosso caso.

O segundo tipo permite a comunicação com outros bancos de dados que tenham suporte à drivers OLE DB, como Oracle, DB2, Access etc. Você pode encontrar na internet drivers OLE DB para uma série de bancos, inclusive Firebird, InterBase, MySQL e muitos outros. No entanto, o uso de OLE DB não é a única opção para acessar bancos de dados a partir de sua aplicação .NET. Muitas empresas fornecem Data Providers nativos para seus bancos, esta é a opção mais otimizada para trabalhar com ADO.NET, pois ninguém conhece melhor o próprio banco do que quem construiu ele, logo o provider nativo será sempre mais rápido.

Cada Managed Provider possui objetos básicos para a comunicação com um servidor de dados, por exemplo: DBConnection, DBCommand, DataReader e DataAdapter. No ADO.NET, seguindo um princípio importante da orientação a objetos, cada objeto tem uma responsabilidade bem definida e coesa; cada um desempenha uma função bem definida. Dessa forma, não se pode, por exemplo, passar um comando SQL em um objeto de conexão, ou usar o mesmo componente para conectar, obter dados, fazer cache e atualizar dados.

A **Tabela1** mostra os principais objetos do grupo Managed Providers. Para que você possa localizar-se mais facilmente é mostrado, para cada objeto/interface, o equivalente na arquitetura dbExpress. Você utilizará os objetos da primeira coluna através de uma das implementações disponíveis (terceira coluna). Por exemplo, no .NET Framework, para usar um DbConnection, você vai usar um SqlConnection no caso do SQL Server; para um DataReader, deve-se usar o SqlDataReader. Os objetos de acesso ao SQL Server estão disponíveis no namespace System.Data.SqlClient.

Ainda na **Tabela 1,** você pode observar a similaridade do ADO.NET com o dbExpress. Cada objeto no ADO.NET implementa uma interface (IDbConnection, IDbCommand, IDataReader, IDataAdapter, IDbTransaction). Essas interfaces são implementadas por cada um dos Managed Providers. De forma semelhante, no dbExpress, cada driver deve implementar as interfaces ISQLConnection, ISQLCommand, ISQLCursor e ISQLDriver. Veja na **Figura 1** a arquitetura básica de Managed Providers do ADO.NET.

Analisando a **Figura 1,** vemos que um DbCommand se conecta a um DbConnection e utiliza um objeto DbTransaction

Objeto/Interface	Função	Implementação no .NET Framework (SQL Server Provider)	No dbExpress
DbConnection/ IDbConnection	Conecta a um banco de dados	SqlConnection	SQLConnection/ISQLConnection
DbCommand/ IDbCommand	Executa um comando SQL no banco	SqlCommnad	(Comandos executados usando-se a interface ISQL-Command)
DataReader/IDataReader	Faz a leitura de dados através de um cursor unidirecional	SqlDataReader	SQLDataSet (usando uma interface ISQLCursor)
DataAdapter/IDataAdapter	Abstrai os objetos DbCommand (e indiretamente um DataReader)	SqlDataAdapter	SQLDataSet+DataSetProvider
DbTransaction/IDbTransaction	Gerencia transações	SqlTransaction	Transações gerenciadas por SQLConnection/ISQL-Connection

Tabela 1. Managed Providers, objetos do .NET Framework e equivalentes no dbExpress do Delphi Win32

para gerenciar a transação com o banco de dados. Um objeto DbCommand não possui equivalência direta no dbExpress, onde os comandos SQL são executados por meio de uma interface ISQLCommand utilizada internamente pelos DataSets unidirecionais, tais como o SQLDataSet. Nesse caso, se você quiser executar uma consulta SQL, ou você a repassa como parâmetro de SQLConnection.Execute ou usa um SQLQuery/SQLDataSet chamando o método ExecSQL. No ADO.NET, isso não é necessário, pois temos um objeto específico.

Um DbCommand pode utilizar dois tipos de execução de comandos SQL, através dos seguintes métodos:

• ExecuteReader: quando o comando for uma consulta (select) que retorna um cursor. Esse método retorna um objeto do tipo IDataReader que você pode utilizar para fazer a leitura dos dados obtidos (unidirecional); é semelhante a chamar Open em um SQLDataSet do dbExpress;

• ExecuteNonQuery: utilize quando o comando for um update, delete ou insert; esse método retorna o número de registros afetados; equivale a chamar um ExecSQL em um SQLDaraSet do dbExpress;

• ExecuteScalar: usado para retornar apenas um objeto, ideal para consultas com funções agregadas que retornam apenas um registro e um único campo.

Existe no ADO.NET um objeto chamado DataAdapter que abstrai o tratamento de quatro objetos DbCommand internos, responsáveis pelos comandos de select, delete, insert e update. Pense em um DataAdapter como um componente Query

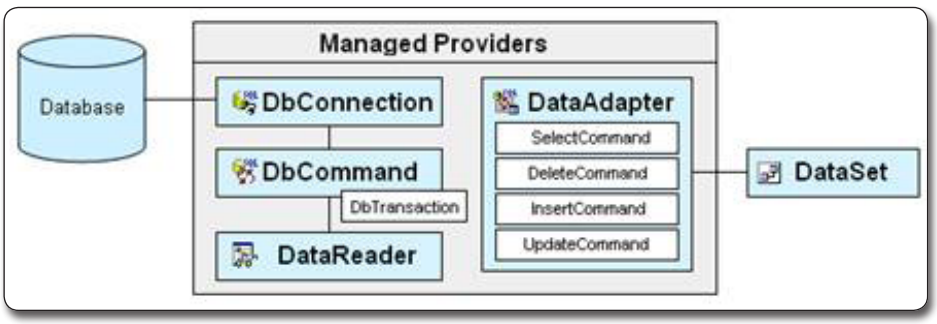


Figura 1. Arquitetura de Managed Providers

no BDE (que faz o select) com um componente Update SQL associado (que gera as instruções de atualização). Já comparado ao dbExpress, o DataAdapter se comporta como um SQLDataSet ou SQLQuery (que faz o select) e um DataSetProvider (responsável pela geração das instruções de atualização).

Diferentemente da dupla SQLDataSet/DataSetProvider, um DataAdapter expõe as instruções de atualização para que você possa alterá-las . No dbExpress, você só consegue isso manipulando ProviderFlags ou manipulando BeforeUpdateRecord, o que é pouco produtivo.

No caso do SelectCommand (que precisa retornar um cursor), o DataAdapter chama seu método ExecuteReader e obtém um objeto DataReader. Um DataReader é responsável pela manipulação do cursor SQL retornado pela execução de um comando; contém vários métodos Get para obtenção de valores de campos (semelhante à interface ISQLCursor do dbExpress).

Um DataReader não faz cache de dados, sua navegação é unidirecional e não é possível alterar seus dados diretamente. Ele possui um método Fill para preencher os dados de outro objeto, chamado DataSet, que é encarregado de fazer a cache. Um DataSet do ADO.NET tem basicamente o mesmo papel de um ClientDataSet do Delphi, com uma estrutura um pouco diferente (veremos detalhes a seguir). Esse objeto faz parte do segundo grupo de objetos do ADO.NET, os componentes de conteúdo (Content Components).

DataSets

O objeto DataSet (no namespace System.Data) tem um importante papel no ADO.NET, seu objetivo é armazenar dados fornecidos por um DataAdapter, fazendo uma cache local, em memória, por meio de objetos DataTable (vistos a seguir).

Um DataSet, juntamente com os objetos DataTable, DataColumn, DataRow, DataView e DataRelation, faz parte do segundo grupo de componentes da arquitetura ADO.NET, os Content Components. Você pode utilizar esses objetos com qualquer um dos Managed Providers fornecidos, como SQL Server.

Um DataSet não armazena dados vindos somente de um banco de dados (trazidos por um DataAdapter). Pode ser utilizado para ler dados de um arquivo XML, de um servidor de aplicações, de uma planilha, ou ainda dados gerados em memória. Esse comportamento é muito similar ao

do ClientDataSet do Delphi. Uma das principais diferenças de um DataSet do ADO.NET para um ClientDataSet é que um DataSet pode fazer cache de mais de uma tabela ao mesmo tempo – ele possui internamente uma coleção de objetos DataTable, que, sem dúvida é o principal objeto do ADO.NET (poderíamos dizer, em um nível mais alto, que um DataSet é uma "coleção de ClientDataSets").

No ClientDataSet, por mais que sua instrução SQL possa trazer dados de várias tabelas usando um join, o que você obtém são dados em formato tubular, que se parecem com uma única tabela do banco. Já um DataSet do ADO.NET se comporta como um "mini banco de dados". Ou seja, ele é capaz de gerenciar várias tabelas, relacionamentos, índices e visões dentro de uma mesma estrutura.

Para preencher um DataSet chamamos o método Fill de um DataAdapter. Como um DataSet pode fazer cache de mais de uma tabela, esse método possui um segundo parâmetro, onde você pode indicar o nome do DataTable interno a ser preenchido. Por isso, ao ligar por exemplo um GridView a um DataSet (através da propriedade DataSource), também é necessário especificar qual o DataTable interno estamos utilizando (propriedade DataMember). Mas isso só no caso do DataSet ter vários DataTables. Isso obviamente é diferente do Delphi Win32, onde um DBGrid aponta apenas para um DataSource. O processo de ligar um componente (como um GridView) a uma fonte de dados é chamado DataBinding no .NET. Veja na **Figura 2** a estrutura de um objeto DataSet.

Sessões

No ASP.NET o objeto Session permite persistir dados na memória do servidor, que estão ligadas a um usuário. Obviamente, cada usuário possui sua própria sessão (não seria nada interessante estarmos comprando produtos em uma loja e vermos produtos de outro carrinho misturados ao nosso). Para fazer essa persistência em memória, usamos o objeto Session.

Lembre-se de que, como estamos usando a memória do servidor, é importante tomar cuidado com cada byte alocado. Por este motivo, existe um timeout (tempo de expiração), que pode ser configurado no Web.Config através do elemento SessionState. No Desktop, temos normalmente GigaBytes de memória para cada usuário, pois cada um está em uma estação de trabalho rodando Windows. Na Web, lembre-se que tudo o que você aloca é no servidor, tudo terá um custo, cada byte é valioso, cada milissegundo é precioso. Ainda mais que agora com a plataforma Windows Azure, você pagará por recursos usados no servidor, então saiba exatamente o que está fazendo no código.

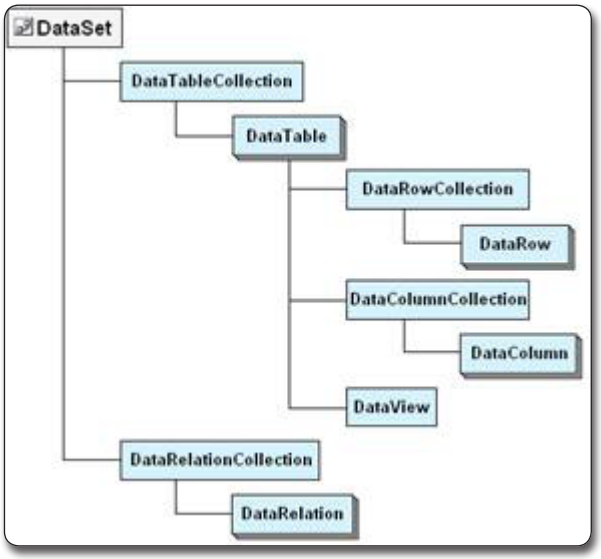


Figura 2. Estrutura do objeto DataSet do ADO.NET

Não temos como comparar o uso de sessões na Web com algo no Desktop, simplesmente porque isso é um recurso que existe para suprir uma necessidade de ambientes state-less como o ASP.NET.

Agora que temos um bom embasamento teórico de tudo o que será mostrado na prática nesta etapa do mini-curso, vamos voltar ao nosso projeto e criar a funcionalidade do carrinho de compras.

Página do Carrinho de Compras

Adicione uma nova página ao Web Site, herdando da Master Page, dando a ela o nome de ShoppingCart.aspx. Coloque no formulário um GridView, um Label e dois ImageButtons. Os botões apontam a propriedade ImageUrl para imagens que estão no disco, dentro de uma pasta no Web Site. Escolha imagens ao seu gosto, uma para representar o fechamento da compra e outra para continuar comprando. Selecione também uma imagem para adicionar o produto ao carrinho (usada a seguir).

No GridView, vamos até a SmartTag adicionar alguns campos do tipo BoundField (Product e Price), ImageField (ProductId) e um ButtonField (Delete). Este GridView será usado, como você já deve estar suspeitando, para exibir os produtos atualmente no carrinho de compras, ou seja, que estão na memória do DataSet. A **Listagem 1** mostra o código ASPX desta página (que pode ser visto em tempo de design na **Figura 3**).

Nota

A partir desta edição, vamos continuar nosso exemplo utilizando o novo Visual Studio 2010, ASP.NET 4.0 e Delphi Prism 2011. Você pode obter uma versão de avaliação dos produtos a partir da sessão Links deste artigo. No entanto, caso opte por utilizar o Visual Studio 2008, poderá continuar a acompanhar o restante do curso, visto que por enquanto não vamos utilizar nenhum recurso exclusivo do .NET Framework 4.0 que não esteja presente no .NET 3.5. Note que, ao abrir o projeto no Visual Studio 2010, o IDE vai perguntar se deseja atualizar os arquivos do projeto para o .NET 4.0, apenas confirme que sim. Também é possível usar o Delphi Prism 2011 no Visual Studio 2008.

Nota

Nota: Se você optar por baixar a última versão deste exemplo a partir do link para download, notará uma pequena mudança. Optei por migrar o banco de dados para o SQL Server 2008 Express, versão que acompanha o Visual Studio 2010. Além disso, ao invés de registrar o banco no servidor através do SQL Server Management Studio, adaptei a aplicação para usar um banco local visível através da pasta App_Data (o que dispensa o registro do banco e torna a configuração e deploy ainda mais simples). Ou seja, para testar este exemplo, basta você baixar e executar.

Code-Behind da página do Carrinho de Compras

A **Listagem 2** mostra o código da página do carrinho de compras. Como você pode ver na **Listagem 1**, alguns controles apontam eventos para métodos que serão implementados neste mesmo arquivo de code-behind.

Listagem 1. Página do Carrinho de Compras - ASPX

```
<%@ Page Language="Oxygene" MasterPageFile="~/MasterPage.master"
AutoEventWireup="true" CodeFile="ShoppingCart.aspx.pas"
Inherits="ShoppingCart" Title="Untitled Page" %>
<asp:Content ID="Content1" ContentPlaceHolderID="ContentPlaceHolder1"
Runat="Server">
<strong>Shopping Cart</strong>
<asp:GridView ID="GridView1" runat="server"
    CellPadding="1" ForeColor="#333333" AutoGenerateColumns="False"
    OnRowCommand="GridView1_RowCommand">
    <FooterStyle BackColor="#5D7B9D" Font-Bold="True" ForeColor="White" />
    <RowStyle BackColor="#F7F6F3" ForeColor="#333333" />
    <EditRowStyle BackColor="#999999" />
    <SelectedRowStyle BackColor="#E2DED6" Font-Bold="True" ForeColor="#333333" />
    <PagerStyle BackColor="#284775" ForeColor="White" HorizontalAlign="Center" />
    <HeaderStyle BackColor="#5D7B9D" Font-Bold="True" ForeColor="White" />
    <AlternatingRowStyle BackColor="White" ForeColor="#284775" />
    <Columns>
      <asp:BoundField DataField="ProductName" HeaderText="Product" />
      <asp:ImageField DataImageUrlField="ProductId"
        DataImageUrlFormatString="~/Products/{0}.jpg">
      </asp:ImageField>
      <asp:BoundField DataField="Price" HeaderText="Price" />
      <asp:ButtonField CommandName="Del" Text="Delete" />
    </Columns>
  </asp:GridView>
    Total =
    <asp:Label ID="Label1" runat="server" Font-Bold="True" Text="Label"></asp:Label>
    <asp:ImageButton ID="ImageButton1" runat="server"
      ImageUrl="~/images/checkout.gif" OnClick="ImageButton1_Click" />
<asp:ImageButton ID="ImageButton2" runat="server"
  ImageUrl="~/images/continue.gif" OnClick="ImageButton2_Click" /><br />
</asp:Content>
```

Listagem 2. Code-Behind da página - PAS

```
namespace;

interface

uses
 System,
 System.Data,
 System.Configuration,
 System.Collections,
 System.Web,
 System.Web.Security,
 System.Web.UI,
 System.Web.UI.WebControls,
 System.Web.UI.WebControls.WebParts,
 System.Web.UI.HtmlControls,
 System.Data.SqlClient;

type

 ShoppingCart = public partial class(System.Web.UI.Page)
 private
   method CreateDataSet(): DataSet;
   method GetDataSet(): DataSet;
   method AddProduct(ProductId: System.Int32);
   method GridDataBind();
 protected
   method Page_Load(sender: System.Object; e: EventArgs);
   method GridView1_RowCommand(sender: System.Object; e: GridViewCommand
    EventArgs);
   method ImageButton1_Click(sender: System.Object; e: ImageClickEventArgs);
   method ImageButton2_Click(sender: System.Object; e: ImageClickEventArgs);
 end;

implementation

method ShoppingCart.CreateDataSet(): DataSet;
begin
 var ds: DataSet := new DataSet();
 var dt: DataTable := ds.Tables.Add();
 dt.Columns.Add('ProductId');
 dt.Columns.Add('ProductName');
 dt.Columns.Add('Price', typeof(System.Double));
 dt.Columns.Add('Total', typeof(System.Double),'sum(Price)');
 exit ds
end;

method ShoppingCart.GetDataSet(): DataSet;
begin
 if Session['ShoppingCart'] = nil then
  Session['ShoppingCart'] := CreateDataSet();
 result := DataSet(Session['ShoppingCart'])
end;

method ShoppingCart.AddProduct(ProductId: System.Int32);
begin
 var con: SqlConnection := new SqlConnection(System.Web.Configuration.
  WebConfigurationManager.ConnectionStrings['LOJA'].ConnectionString);
 var SQL: System.String :='select ProductName, Price from Products ' +'where
 ProductId=' + ProductId.ToString();
 var cmd: SqlCommand := new SqlCommand(SQL, con);
 con.Open();
 var dr: SqlDataReader := cmd.ExecuteReader(CommandBehavior.CloseConnection);
 dr.Read();
 var ds: DataSet := GetDataSet();
 var row: DataRow := ds.Tables[0].NewRow();
 row['ProductId'] := ProductId.ToString();
 row['ProductName'] := dr['ProductName'].ToString();
 row['Price'] := dr['Price'].ToString();
 ds.Tables[0].Rows.Add(row);
 dr.Close()
end;

method ShoppingCart.Page_Load(sender: System.Object; e: EventArgs);
begin
 if not IsPostBack then
 begin
  if Request.QueryString['ProductId'] <> nil then
   AddProduct(System.Int32.Parse(Request.QueryString['ProductId']));
  GridDataBind()
 end
end;


method ShoppingCart.GridDataBind();
begin
 GridView1.DataSource := GetDataSet();
 GridView1.DataBind();
 if GetDataSet().Tables[0].Rows.Count > 0 then
  Label1.Text := GetDataSet().Tables[0].Rows[0]['Total'].ToString()
end;

method ShoppingCart.GridView1_RowCommand(sender: System.Object; e: Grid-
ViewCommandEventArgs);
begin
 if e.CommandName = 'Del' then
 begin
```

Continuação: Listagem 2. Code-Behind da página - PAS

```
  GetDataSet().Tables[0].Rows.RemoveAt(System.Int32.Parse(e.CommandArgument.
   ToString()));
  GridDataBind()
 end
end;

method ShoppingCart.ImageButton2_Click(sender: System.Object; e: ImageClick
 EventArgs);
begin
 Response.Redirect('~/Home.aspx')
end;

method ShoppingCart.ImageButton1_Click(sender: System.Object; e: ImageClick
 EventArgs);
begin
 Response.Redirect('~/Login.aspx')
end;

end.
```

O método CreateDataSet cria um objeto do tipo DataSet, manualmente via código (assim como criamos tabelas temporárias em memória usando ClientDataSets). Colocamos uma coluna para cada tipo de dado que vamos armazenar na sessão para o carrinho de compras. Note algo em especial, o campo Total é semelhante a um Aggregate do ClientDataSet, ele possui uma função SUM que soma todos os subtotais da compra.

O método GetDataSet é simplesmente um "auxiliar" para acesso ao DataSet. Ele faz um teste, se o DataSet já foi criado, ou seja, está em sessão na memória, então este objeto é retornado, caso contrário, ele é criado usando o método apresentado no parágrafo anterior.

O método AddProduct é um dos mais importantes da página, ele busca por informações de um dado produto no banco de dados e armazena essas informações no DataSet. Veja que para isso usamos objetos do grupo de classes conectadas do ADO.NET, como conexões e DataReaders. Veja neste mesmo código como fazemos para ler uma string de conexão via código, a partir do Web.Config.

O Page_Load contém pouco código, porém fundamental parar o funcionamento da página. A página que exibe os detalhes do produto vai passar para o carrinho de compras apenas o ID do produto que foi escolhido, ou seja, adicionado ao carrinho (faremos isso a seguir). O ID é passado na própria URL, de forma que capturamos este código e repassamos ao método explicado no parágrafo anterior.

O GridDataBind não faz muita coisa além de configurar o DataSource do Grid para apontar para o DataSet de memória e mostrar o total da compra no Label. O RowCommand testa se o usuário clicou no Delete, para retirar o produto do carrinho. Os demais métodos são os responsáveis por responder aos Clicks nos ImageButtons, redirecionando o usuário para as respectivas páginas.

Listagem 3. Adicionando um produto ao carrinho

```
method ShowProduct.ImageButton1_Click(
sender: System.Object;
e: System.Web.UI.ImageClickEventArgs);
begin
  Response.Redirect("~/ShoppingCart.aspx?ProductId=" +
   Request.QueryString["ProductId"]);
end;

method ShowProduct.ImageButton2_Click(
sender: System.Object;
e: System.Web.UI.ImageClickEventArgs);
begin
 Response.Redirect("~/Home.aspx");
end;
```

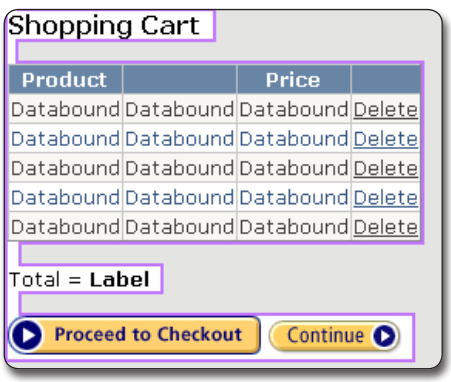


Figura 3. Página do carrinho de compras, em design time

Figura 4. Botões para adicionar produto ao carrinho e continuar a compra

Figura 5. Carrinho de compras em funcionamento

Adicionando um produto ao carrinho

O último passo é permitir que o usuário adicione o produto que está visualizando ao carrinho, na página ShowProduct.aspx. Para isso, usamos dois ImageButtons, como você pode ver na **Figura 4**. O código dos botões pode ser visto na **Listagem 3**. Repare que ao clicar no botão de Add, chamamos a página que criamos anteriormente (ShoppingCart.aspx) e passamos o ID do produto selecionado.

Testando

A **Figura 5** mostra o exemplo em execução, onde escolhemos alguns produtos,

visualizamos seus dados, imagem, preço, e a seguir adicionamos ao carrinho de compras (que combina o DataSet em sessão e o GridView).

Conclusão

Isso finaliza esta etapa do mini-curso. Somente agora, na quinta parte, adicionamos as primeiras linhas de código em Delphi Prism! Tudo bem que, como gosto de dizer, o que usamos é um "D#", visto que o código é uma mescla de C# com Delphi Win32. A sintaxe é do Pascal, com aditivos que temos no C# 4.0, porém todas as classes que usamos são do .NET Framework, aqui não existe VCL (nem mesmo a falecida VCL.NET). Porém, procurei sempre apresentar cada recurso comparando ao nosso tradicional Delphi Win32, pois o objetivo deste curso é facilitar o aprendizado da tecnologia através de comparativos. Um grande abraço e até a próxima edição.

Guinther Pauli
guinther@devmedia.com.br
guinther.pauli@gmail.com
http://twitter.com/guintherpauli
http://gpauli.com

Desenvolve em Delphi há 15 anos, tendo utilizado a ferramenta desde a versão 1, passando pelo Turbo Pascal e Borland Pascal for Windows. Certificado em Delphi 3, 5, 6, 7, 2005, 2006, Delphi para Web e Delphi para Linux, palestrante em 5 edições da Borland Conference. Certificado Microsoft MCP, MCAD, MCSD.NET, MCTS e MCPD (C#, ASP.NET), utilizando a plataforma .NET desde a versão beta. Editor das Revistas ClubeDelphi, .NET Magazine Brasil e EASY .NET Magazine, professor de Pós-Graduação da UNIVEM-SP, na área .NET. Especialista em desenvolvimento Delphi, C#, .NET, ASP.NET, Visual Studio, atuando também em projetos de migração, orientando empresas e desenvolvedores na adoção de melhores práticas, ministra treinamentos em .NET focados em equipes Delphi. Foi arquiteto e desenvolvedor ASP.NET do site de doações on-line do projeto Criança Esperança da Rede Globo / Unesco.

Site oficial do Delphi Prism
http://www.embarcadero.com/products/delphi-prism

Wiki do Delphi Prism
http://prismwiki.codegear.com/en/Main_Page

Download do Delphi Prism
https://downloads.embarcadero.com/free/delphi_prism

Site oficial do ASP.NET
http://www.asp.net

Visual Studio
www.microsoft.com/visualstudio

Blogs do autor
http://twitter.com/guintherpauli
http://gpauli.com

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Faça um *up grade* em sua carreira.

Em um mercado cada vez mais focado em qualidade ter conhecimentos aprofundados sobre requisitos, metodologia, análises, testes, entre outros, pode ser a diferença entre conquistar ou não uma boa posição profissional. Sabendo disso a DevMedia lança para os desenvolvedores brasileiros sua primeira revista digital totalmente especializada em Engenharia de Software. Todos os meses você irá encontrar artigos sobre Metodologias Ageis; Metodologias tradicionais (*document driven*); ALM (*application lifecycle management*); SOA (aplicações orientadas a serviços); Analise de sistemas; modelagem; Métricas; orientação à objetos; UML; testes e muito mais. **Assine já!**

DevMedia group

Chegou o Sistema de Créditos DevMedia

Agora o **conteúdo completo** do nosso site está **ao seu alcance!**

A partir de agora todas as **4000 vídeo-aulas** do site DevMedia podem ser compradas individualmente.

Economia - Você não precisa mais assinar oito produtos diferentes para ter acesso ao conteúdo completo do site!

Todas as vídeos que você sempre quis ver a partir **R$ 0,99!**

Saiba mais sobre o Sistema de Créditos!

DevMedia GROUP

www.devmedia.com.br/creditos

Hospede seus sites de forma simples, rápida e ilimitada.

Plano **Básico**

- 1 domínio principal
- 50 domínios hospedados
- Transferência **ilimitada**
- Espaço **ilimitado**
- 10 contas de e-mail
- PHP, ASP e ASP.NET
- Painel de controle Plesk
- Suporte 24hs

Tudo isso por apenas

R$ **15,90** por mês
pagamento trimestral

Assine agora mesmo e descubra por que Hospedix é a hospedagem de sites preferida entre os profissionais.

- Conheça também nossos planos profissionais com registro de domínio grátis e isenção de taxas anuais enquanto for cliente.

HOSPEDIX
HOSPEDAGEM PROFISSIONAL
www.hospedix.com.br

30 DIAS GRÁTIS

Ao assinar, digite o código de desconto exclusivo para leitores desta revista e ganhe 30 dias inteiramente grátis.

RVSTMEDIA

As promoções anunciadas não são cumulativas com outras promoções e são válidas apenas para clientes Hospedix ativos.